# Doppler Channel Emulation of High-Bandwidth Signals

by

Joseph William Colosimo

B.S., Massachusetts Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering
in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013



Author .....
Department of Electrical Engineering and Computer Science
Jun 1, 2013

Certified by .....
Aradhana Narula-Tam
Assistant Group Leader; MIT Lincoln Laboratory
VI-A Company Thesis Supervisor

Certified by .....
Muriel Medard
Professor of Electrical Engineering and Computer Science
MIT Thesis Supervisor

Accepted by .....
Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

**Doppler Channel Emulation of High-Bandwidth Signals**

by

Joseph William Colosimo

Submitted to the Department of Electrical Engineering and Computer Science
on Jun 1, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering
in Electrical Engineering and Computer Science

## Abstract

The Airborne Networks Group at MIT Lincoln Laboratory has funded the construction of a channel emulator capable of applying, in real-time, environmental models to communications equipment in order to test the robustness of new wireless communications algorithms in development. Specific design goals for the new emulator included support for higher bandwidth capabilities than commercial channel emulators and the creation of a flexible framework for future implementation of more complex channel models. Following construction of the emulator's framework, a module capable of applying Doppler shifting to the input signal was created and tested using DVB-S2 satellite modems. Testing not only verified the functionality of the emulator but also showed that DVB-S2 modems are unequipped to handle the continuous spectral frequency shifts due to the Doppler effect. The emulator framework has considerable room for growth, both in terms of implementing new channel transformation models as well as the re-implementation of the emulator on custom hardware for emulation of channels with wider bandwidths, more complex noise sources, or platform-dependent spatial blockage effects.

Thesis Supervisor: Aradhana Narula-Tam
Title: Assistant Group Leader; MIT Lincoln Laboratory

Thesis Supervisor: Muriel Medard
Title: Professor of Electrical Engineering and Computer Science

# Acknowledgments

MIT's VI-A program[1] made this thesis possible. Through VI-A, MIT Lincoln Laboratory gave me access to a wealth of amazing people and resources. My sincere thanks to Kathy Sullivan for her work to keep VI-A running smoothly.

Group 65 leader Matt Kercher was my supervisor, mentor, and friend. He was also always happy to provide whatever resources he could to aide in the completion of this project that he had originally envisioned. His guidance was indispensable.

I'm enormously thankful for Chayil Timmerman's help. For nearly a year, he mentored me in the art of digital design in timing-constrained systems; I learned much. He even let me steal his lab equipment for extended periods of time.

I would also like to thank Aradhana Narula-Tam, assistant group leader, for her advice, as well as group administrator Evelyn Bennett, who made sure I always had the resources I needed. My thanks to Larry Bressler and Ed Kuczynski as well.

Professor Muriel Medard graciously agreed to become my faculty thesis advisor very late in the game. My department advisor, Jeff Shapiro, was always kind to his advisees throughout our years at MIT. I also had the great fortune of being able to spend some time with Professor Jim Roberge, the VI-A liaison to Lincoln Lab.

The initial seeds of interest in digital design were planted by Roberto Acosta at BAE Systems and Gim Hom at MIT. In that way, they have played a major role in where I have chosen to direct my interests and time.

My friends have been a continuing source of both healthy distraction and motivation. 3pm coffee breaks with Karen Sittig often got me through frustrating days. I am deeply thankful to Qiaodan Jin Stone for her continued support.

Finally, I am eternally indebted to my family, especially my parents, who made great sacrifices to give me opportunities they never had. I am truly thankful each day for their support and love. I can only hope that I have done their hard work a modicum of justice.

[1] http://vi-a.mit.edu

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Fundamentally, a channel emulator takes an input signal, transforms it in a manner that, as closely as possible, mimics the environmental effects of noise, fading, or signal path impairments, and transmits the newly-transformed signal. For example, designers of wireless consumer products often employ channel emulation to determine how well their new devices perform in sub-optimal environments such as in buildings or tunnels. Channel emulation allows them to test their devices without leaving the lab. Figure 1-1 describes a typical channel emulation setup.

Because a channel emulator operates on a signal in real-time, it is especially useful in testing the *end-to-end* functionality of a given target. For example, when designing a video communication device, one might hook two prototype devices up to a channel emulator, instruct the emulator to apply various sub-optimal environments, and directly see how well all of the components of the devices deal with this damaging channel: from the hardware to the networking stack to the video codec itself.

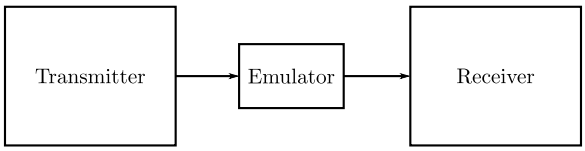


Figure 1-1: **Channel emulation in a nutshell**. A channel emulator sits between a transmitter and a receiver. In real-time, it modifies the transmitter's signal in some controllable way and presents the transformed signal to the receiver.

## 1.1 Comparison to Simulation

Traditional simulation techniques and channel emulation try to solve different problems.

Simulating a portion of a system as it transmits data through a lossy channel can provide very accurate and repeatable results. However, it also generally requires constructing a complete and accurate model of the system under test. Therefore, typically only small portions of the entire system are modeled and simulated at a given time. If the proposed test involves a system that is too large, it would not only take a considerable effort to build an appropriate simulation, it could also be very computationally complex, resulting in very long run times.

Again referring to the video communication example, an engineer might build a model of the video transmission algorithm and simulate that part as one portion of the system. Another may build a model of the upper portions of the underlying networking stack and simulate packets of data sent through this protocol layer. Yet another might build a model for the lower portions of that network stack and investigate signal integrity in different conditions.

Channel emulation provides *end-to-end* testing capability so that it is not necessary to build individual models of the testing system. Instead, the system itself is used directly in testing. Problems related to the complexity of the system — whether it, for example, has algorithms that are based on real-time clocks, or requires extremely high sampling frequencies that do not lend themselves to prolonged periods of data capture for later analysis — are completely orthogonal to the construction of a channel emulator.

Channel emulators face their own set of design challenges. The root of these problems it that a channel emulator must be able to operate on a streaming set of input data. That is, it must have a throughput ratio of 1. If this is not the case, the amount of time a single emulation can run becomes finite. Thus sacrifices may be necessary, such as reducing the complexity of models the channel emulator can implement. Channel emulators also typically require fairly powerful hardware,

usually at least as powerful as the devices they test.

When designing a system, it is not necessarily better to choose emulation over simulation or vice-versa; it is usually better to choose both.

## 1.2 Constructing a Channel Emulator

There are two overarching significant challenges in the field of channel emulation:

- Research of a given environment and development of the mathematics to match that environment's transformation.
- Implementation of these (often complex) mathematics into some kind of device capable of performing them in real-time.

The bulk of this thesis deals with the latter problem as it appears in the context of high-bandwidth aircraft-to-aircraft communication in a military setting. Specifically, we developed a channel emulator capable of applying in real-time the phenomenon known as Doppler shifting to a high-bandwidth signal.

The overarching goal was to build a *framework* for constructing future, more complicated channel emulators capable of modeling much more complex scenarios such as multipath fading, additive noise, and interference. To that end, the implementation of a Doppler emulation "module" served as a suitable test of the emulator framework's overall functionality and extendability.

This channel emulator was specifically designed to operate on waveforms with bandwidths of approximately 100 MHz or less. Commercial channel emulators typically do not work with such wideband signals. The center frequency of the signal used to test the initial emulator framework implementation was 1.25 GHz, but in practice, the channel emulator can be modified to operate on a spectrum with a different center frequency.

With regard to the Doppler module, the emulator was designed to shift a spectrum by up to ±10 KHz. Back-of-the-envelope calculations using math in Chapter 2, particularly Equation 2.7, indicate that this is a very generous upper bound, as frequency

shifts for most real-world scenarios are almost always below 4 KHz. In actuality, the resulting architecture allowed the frequency to be shifted by multiple MHz, as shown in Section 3.6.5.

Finally, when not applying Doppler shifting or any other signal transformation, the channel emulator should essentially look like wire. With regard to delay, the channel emulator itself should not induce a signal delay that would appear significant to two aircraft 100 m apart. As shown in Chapter 3, the overall signal delay is around 20 clock cycles at 1 GHz, so the corresponding delay is simply $c \cdot \frac{20}{1 \cdot 10^9} = 5.82\,\text{m}$, where $c$ is the speed of light in air. This latency overhead is therefore extremely small.

With regard to fidelity, the signal-to-noise ratio ($SNR$) of a signal as it passes through the channel emulator should be close to that of a signal passing through a wire. Generally, for this kind of application, 60 dB is generally considered to be superb. Testing showed that the actual SNR for a given test spectrum was closer to around 50 dB, which is still excellent.

### 1.2.1 Hardware Decisions

One of the first steps in developing this project was the research and evaluation of hardware on which to implement the channel emulator.

Xilinx's Virtex 6 FPGA was chosen to be the fundamental processing unit for the channel emulator. Section 4.1 details the justification for choosing to emulate environmental effects digitally with an FPGA and Section 4.1.1 specifically justifies the choice of this particular FPGA family with respect to its hardware capabilities.

Because the interface between the channel emulator and the testing equipment is RF, a intermediate processing stage whereby analog signals are converted to digital and vice-versa is necessary. To accomplish this, an interface containing analog-to-digital converters ($ADCs$) and digital-to-analog converters ($DACs$), described in 4.1.2, was used. Both the ADCs and the DACs sampled at 1 GSPS, one billion samples per second. As a consequence, high-speed, wideband signals could be processed with the channel emulator.

## 1.3 Notation

The following notation is used throughout the thesis.

### Symbolic Definitions

An equation describes how some set of symbols interact with one another. To clarify what those symbols mean, a table may be placed below the equation, like so:

$$c^2 = a^2 + b^2 \tag{1.1}$$

| | |
|---|---|
| c | hypotenuse |
| a | one side |
| b | other side |

### Time-Ordered Sequence

For a chronological sequence where $x_1$ comes after $x_0$, $x_2$ comes after $x_1$, etc., the following notation is used:

$$\llcorner x_0 \to x_1 \to x_2 \to \cdots \urcorner$$

### Complex Numbers

A complex sample is denoted as $\langle R, I \rangle$ and is equivalent to the mathematical notation $R + jI$.

### Buses

A wire bus is represented by:

$$\langle\!\langle \ \mathtt{A} \cdot \mathtt{B} \cdot \mathtt{C} \ \rangle\!\rangle$$

The most-significant bits are on the left and least-significant are on the right. The operator $\cdot$ concatenates successive bits. For example:

$\langle\!\langle$ `0xAA · 0x55` $\rangle\!\rangle$

is equivalent to the number 0xAA55, or as an unsigned integer, 43605.

## Array-Like Data Structures

Array-like data structures are represented as matrices. For example, consider the data structure described by Equation 1.2:

$$X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{1.2}$$

In this example, $X[0][0] = a$, $X[1][0] = d$, and $X[0][1] = b$.

# Chapter 2

# Understanding the Doppler Effect

The phenomenon known as the Doppler effect is commonly encountered in daily life.

For example, consider the scenario where an observer is standing still while a car blaring its horn drives past him at constant speed. As the vehicle approaches, the horn's pitch appears to be higher than if the car were still and when the car passes the observer, it appears to be lower.

The sound wave is first being compressed from the observer's perspective as the vehicle approaches. It then appears "stretched" as the vehicle drives away. The amount that the wave appears stretched or compressed is linearly related to the relative speed of the source/observer pair, the frequency of the wave, and the speed of the wave in the medium. Figure 2-1 demonstrates the apparent effect.

In this chapter, we develop a model for applying the Doppler effect to RF spectra.

## 2.1 Shifting a Sine Wave

Because the Doppler effect is dependent on the frequency of the waveform being shifted, it is easiest to begin by considering the effect on a single sine wave.

The vehicle scenario described above illustrates the subsonic Doppler effect: the case where the relative speed of the source and destination is less than the speed of the wave. In the scenario of this thesis, we are dealing with RF transmission, where the speed of the wave is the speed of light in air (around $3{\cdot}10^8$ m/s) and the

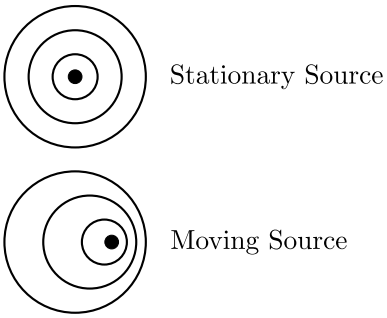


Figure 2-1: **Doppler shifting example**. When the source is stationary, the wave incident on an observer is some frequency, as shown in the upper example. When the object is moving to the right , the apparent wave frequency is different based on where the observer is in relationship to the source. If the observer is directly to the right of the source, it will see an apparently frequency higher than if the source was stationary. If it were directly to the left of the source, it would see a lower frequency.

relative source-destination speed is around 600 m/s. Clearly, this falls under the same scenario.

In general, for some sine wave at frequency $f$ in a one-dimensional world, Doppler [1] observed that

$$f' = \left(\frac{v_d + v_w}{v_s + v_w}\right) f \tag{2.1}$$

| | |
|---|---|
| $f'$ | destination's perceived wave frequency |
| $v_w$ | speed of the wave in the current medium |
| $v_d$ | speed of the signal's receiver |
| $v_s$ | speed of the signal source |
| $f$ | original frequency of the wave |

However, $v_d \ll v_w$ and $v_s \ll v_w$. We can simplify this equation in the following manner.

$$f' = \left(\frac{v_d + v_w}{v_s + v_w}\right) f \tag{2.2}$$

$$= \left(\frac{v_d}{v_s + v_w} + \frac{v_w}{v_s + v_w}\right) f \tag{2.3}$$

$$\approx \left(\frac{v_d - v_s}{v_s + v_w - v_s} + \frac{v_w + v_s}{v_s + v_w}\right) f \tag{2.4}$$

$$\approx \left(\frac{v_d - v_s}{v_w} + 1\right) f \tag{2.5}$$

$$\approx \left(\frac{v_r}{v_w} + 1\right) f \tag{2.6}$$

| | |
|---|---|
| $v_r$ | relative speed between the receiver and the source (i.e. $v_d - v_s$) |

Thus, if we are interested in the frequency *change*, that is, the difference between the actual signal frequency and its perceived frequency as measured by the receiver, we can describe $\Delta f$ as

$$\Delta f = f' - f = \frac{v_r}{v_w} f \tag{2.7}$$

## 2.2 Shifting a Spectrum

Given Equation 2.7, we can now consider the effects of the Doppler shift on a spectrum of frequencies, i.e. a collection of sine waves. Consider the left spectrum in Figure 2-2. Because of the frequency dependency in the equation, each component of the spectrum will change slightly differently. For example, if one were to apply a positive $v_r$, the result would be represented by spectrum drawn to the right in the same figure. The lowest frequency component of the spectrum will shift less than the highest frequency component. The resulting effect would be that the spectrum appears to spread out, as shown in the figure.

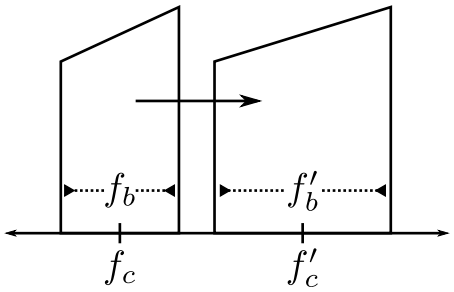


Figure 2-2: **Doppler shift of a spectrum**. The left spectrum is shifted upward to become the right spectrum. The spectrum's parameters change appropriately.

## 2.3 Intelligently Representing a Spectrum Shift

In communications, a transmitted signal is often coarsely characterized by its *carrier* (or *center*) *frequency* and its *bandwidth*. For example, referring again to 2-2, one can easily see that the original carrier frequency of the signal is $f_c$ and the bandwidth is $f_b$.

Given this representation of a signal, it makes sense to represent the frequency shift due to the Doppler effect in an analogous fashion. That is, one can deconstruct the shift of a given frequency into a change in the carrier frequency and a change in a given component.

Consider a spectrum with carrier frequency $f_c$. Consider a component that is $f_x + f_c$. Then, using Equation 2.7, we can decompose $\Delta f$ into two components as follows:

$$\Delta f = \frac{v_r}{v_w}(f_c + f_x) \tag{2.8}$$

$$= \frac{v_r}{v_w} f_c + \frac{v_r}{v_w} f_x \tag{2.9}$$

$$= \Delta f_c + \Delta f_x \tag{2.10}$$

| | |
|---|---|
| $\Delta f_c$ | carrier shift |
| $\Delta f_x$ | component shift |

Thus, the component $\Delta f_c$ is shared among all components in the spectrum. $\Delta f_x$

is unique to each component.

Therefore, we can represent a Doppler shift on a spectrum with two components: an additive frequency shift applied to the whole spectrum ($\Delta f_c$) and individual corrective shifts ($\Delta f_x$ for each $f_x$ in the spectrum).

## 2.4 Determining the Need for the Corrective Shift

The corrective shift component is smaller than the additive shift because for any given component $f_x$ in the spectrum, $f_x \leq \frac{f_b}{2}$, where $f_b$ is the spectrum bandwidth.

How important is this corrective shift? Consider the outermost frequency in a spectrum, namely $f_c + \frac{f_b}{2}$. We can find the error between the full frequency shift and an approximated shift that is just the center frequency component. Let $f_x = \frac{f_b}{2}$.

$$\text{error} = \left| \frac{\Delta f_{\text{full}} - \Delta f_{\text{approx}}}{\Delta f_{\text{full}}} \right| \tag{2.11}$$

$$= \left| \frac{(\Delta f_c + \Delta f_x) - \Delta f_c}{\Delta f_c + \Delta f_x} \right| \tag{2.12}$$

$$= \left| \frac{\Delta f_x}{\Delta f_c + \Delta f_x} \right| \tag{2.13}$$

$$= \left| \frac{\frac{v_r}{v_w} f_x}{\frac{v_r}{v_w} (f_c + f_x)} \right| \tag{2.14}$$

$$= \left| \frac{f_x}{f_c + f_x} \right| \tag{2.15}$$

$$= \frac{\frac{f_b}{2}}{f_c + \frac{f_b}{2}} \tag{2.16}$$

$$= \frac{f_b}{2f_c + f_b} \tag{2.17}$$

Therefore, the error does not depend on the relative velocity or characteristics of the wave speed; it only depends on the carrier frequency and the bandwidth. In the scenario of this thesis, $f_c = 1.25\,\text{GHz}$ and $f_b = 100\,\text{MHz}$, yielding an error of 3.84%.

We can also determine just how much the outermost frequency component will be approximated if we know $v_r$ and $v_w$. That value is simply:

$$f_{\text{error}} = \Delta f_c + \Delta f_x - \Delta f_c \tag{2.18}$$

$$= \frac{v_r}{v_w} f_x \tag{2.19}$$

$$= \frac{v_r}{v_w} \frac{f_b}{2} \tag{2.20}$$

## 2.5 Defining $v_r$ in the Real World

We have characterized $\Delta f$, but have neglected to describe exactly what $v_r$ represents.

When a wave propagates in the direction from the transmitter to the receiver, the Doppler effect causes compression or dilation along the axis of propagation. The relevant component of the relative velocity between the transmitter and the receiver is also along this axis.

In this thesis, we will consider the relative position and velocity of the transmitter and receiver as vectors in the Cartesian coordinate space. The relevant vectors in the transmitter-receiver system are $\vec{r}_{sd}$, the relative position between the source and the destination, and $\vec{v}_{sd}$, the relative velocity between the source and destination.

Therefore, we can define $v_r$ as follows:

$$v_r = -\text{proj}^{\text{sc}}_{\vec{r}_{sd}} \vec{v}_{sd} \tag{2.21}$$

Recall that in Cartesian space, the scalar projection of a vector $\vec{a}$ onto a vector $\vec{b}$ is:

$$\text{proj}^{\text{sc}}_{\vec{b}} \vec{a} = \vec{a} \cdot \hat{b} \tag{2.22}$$

$$= \frac{\vec{a} \cdot \vec{b}}{|\vec{b}|} \tag{2.23}$$

We call $v_r$ the *radial velocity* because it is the component of the velocity along the

relative position axis. For example, if the source was traveling around the destination in a circular motion, the radial velocity would be zero and the perceived shift in frequency would be 0.

# Chapter 3

# Signal Processing Architecture

A channel emulator should appear virtually transparent save for the actual signal transformation for emulating the environment. Therefore, it is important to develop an architecture that modifies the input signal with very low latency and high resolution.

This chapter discusses the overall architecture that this channel emulator employs to apply Doppler shifting to an input signal. Recall that the relevant design constraints dictate that the input carrier frequency is 1.25 GHz and the bandwidth limit is 100 MHz, which is nearly twice that of most commercial channel emulators.

As discussed in Section 4.2.1, due to clocking restraints, all data that is produced from the FPGA's interface to the ADC as well as the data that enters the FPGA's interface to the DAC is parallelized into sets of 8 samples per system clock, with the most significant set of bits representing the latest sample, $S_7$, and the least significant representing the oldest, $S_0$. We call this bus an *octet*.[1]. Data passed between each stage of the signal processing architecture looks like:

$$\langle\!\langle \; S_7 \cdot S_6 \cdot S_5 \cdot S_4 \cdot S_3 \cdot S_2 \cdot S_1 \cdot S_0 \; \rangle\!\rangle$$

The signal processing chain is described in Figure 3-1. An anti-aliased signal is presented to the input ADC. The signal is downconverted to baseband and digitally

[1]In this thesis, the term *octet* refers to a set of 8 samples concatenated together into a single bus. The samples can be of any size. Here, *octet* does not refer to an 8-bit value as it traditionally does in computing.

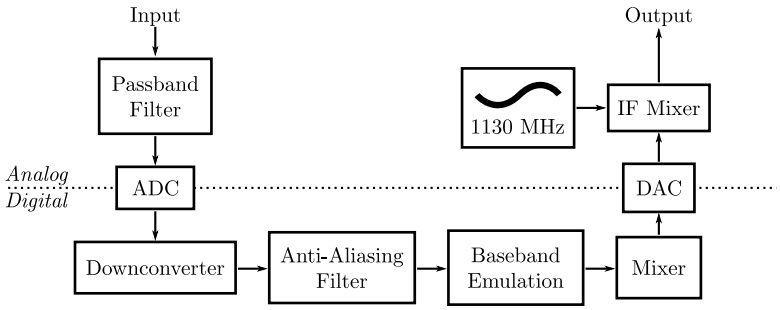


Figure 3-1: **Signal chain overview**. Illustration of the signal processing chain, with separation of the analog and digital realms

anti-aliased. Baseband emulations can be applied to the signal at this point. Then, signal is mixed to an intermediate frequency ($IF$) using a frequency-adjustable mixer. The Doppler frequency shift $\Delta f_c$ is applied at the digital mixer stage. The signal is then sent out to the DAC and mixed up to the final center frequency using an analog mixer.

The microarchitecture of each step of the signal processing chain must be designed to handle the octet-at-a-time constraint.

## 3.1 Analog Pre-filtering

The only relevant portion of the incoming signal is the 100 MHz spectrum around the carrier frequency. It is therefore necessary to install an analog passband filter before the ADC to reject anti-aliasing effects from outside the spectrum. The K&L Microwave filter had a passband from 1 GHz to 1.5 GHz, which was suitable for this project because the ADC (as described in Section 3.2) had a sampling frequency of 1 GSPS, so this filter restricted the input of the ADC to a single Nyquist fold.

## 3.2 ADC

The ADC chosen was the Texas Instruments ADS5400. It provides 12-bit samples at 1 billion samples per second (1 GSPS). It also has adjustable gain between 1.52 $V_{pp}$ and 2.0 $V_{pp}$. Sampling theory and, in particular, rules regarding the Nyquist frequency

tell us that a signal spectrum must be within $0.5 \times f_s$ intervals where $f_s$ is the sampling frequency of the ADC. That is, if $f_s$ is 1 GSPS, or 1 GHz, a spectrum that falls between 0 Hz and 500 MHz would be in the first Nyquist fold, between 500 MHz and 1 GHz would be in the second, 1 GHz and 1.5 GHz would be the third, etc.

When an ADC samples a spectrum that is not in the first fold, the spectrum will be folded down into that first region when it appears at the ADC output. For example, the ADC would show a 1.25 GHz component as 250 MHz. If the original region was even (i.e. second fold, fourth fold, etc.), the spectrum will be inverted as it folds down (e.g. 750 MHz would appear as 250 MHz).

In this case, the ADS5400 operates extremely well in the third Nyquist fold. A signal generator was plugged into the analog input of the ADC. It was then swept through several frequencies in the third fold and their corresponding frequencies in the first fold and a few seconds of samples were recorded from the ADC. A Welch Power Spectral Density estimate was then performed on each capture and the SNR between each corresponding pair was compared. The result was that a signal in the third fold had only around a 10dB loss compared to its corresponding signal in the first fold. This was deemed extremely acceptable, as the SNR was still very large – orders of magnitude larger than the loss experienced in a typical wireless environment.

The output of the ADC is a non-inverted spectrum centered at 250 MHz.

## 3.3 Downconversion

In the first digital processing stage, the incoming signal is downconverted to baseband, yielding I and Q outputs. The process of downconversion usually involves multiplying the input signal by a cosine and also by a sine at the same frequency. The first result is the in-phase component and the second is the quadrature phase component.

There is a slight optimization that can be taken when downconverting a signal with carrier frequency $\frac{f_s}{4}$ where $f_s$ is the sampling frequency. Namely, the incoming data samples can be multiplied by 1, $i$, $-1$, and $-i$ in that order to generate I and Q. This technique was employed in the implementation of the down-converter, but

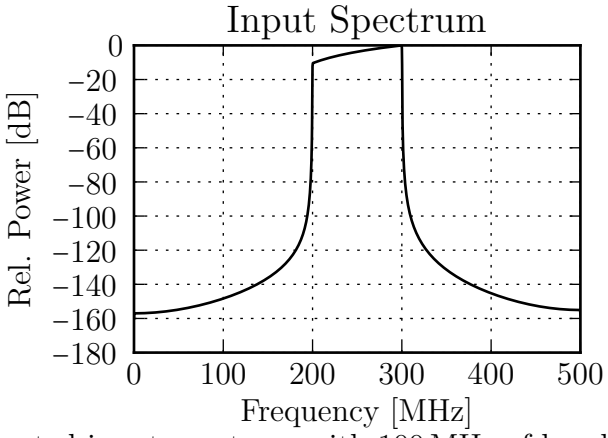

Figure 3-2: Generated input spectrum with 100 MHz of bandwidth, centered around 250 MHz, created by summing and normalizing a series of sine waves with increasing frequencies and amplitudes.

the FPGA has enough resources for a full DDC if a different carrier frequency is ever chosen for the emulator to operate on.

Figure 3-2 shows a test spectrum applied to the DDC and Figure 3-3 shows the I and Q outputs that are the result of the DDC. The latter figure shows two plots: the upper is the result of applying the operation with floating-point precision. The lower is the result of applying the operation using the hardware simulator Modelsim with a fixed-point hardware implementation of the design. Figures 3-6 and 3-10 will also show two plots each in a similar manner. Comparing the two allows us to see the effect of the lower-resolution fixed-point numeric format on the generated signals. Section 3.9 discusses choices for numeric representation in the hardware design.

Implementing this design on the set of parallel input samples is relatively straightforward. For an input $S_0$ through $S_7$, the final output looks like the following sequence

$$\llcorner S_0 \rightarrow jS_1 \rightarrow -S_2 \rightarrow -jS_3 \rightarrow S_4 \rightarrow jS_5 \rightarrow -S_6 \rightarrow -jS_7 \urcorner \tag{3.1}$$

To generate this set of samples, the downconversion module creates two octets, one for in-phase components and one for quadrature components. It computes $-S_2$, $-S_3$, $-S_6$, and $-S_7$ by taking the complement of each sample and adding 1 (because the samples are in two's complement format). Finally, it rearranges the set of input samples and negated samples to produce the output buses:

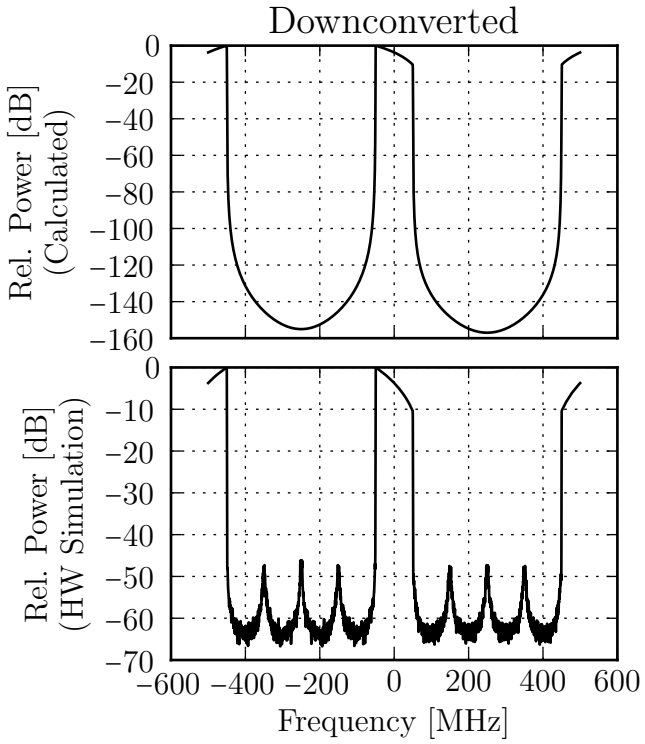


Figure 3-3: Result of applying an $f_s/4$ downconversion to the input spectrum in Figure 3-2. The baseband signal is now represented by I and Q samples. It also has an image as a result of aliasing. This will need to be removed with a low-pass filter.

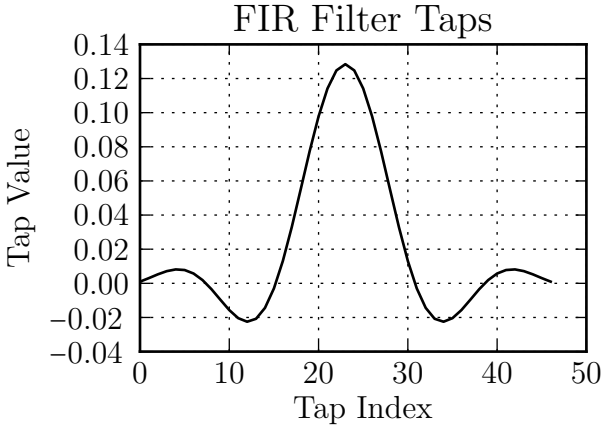

Figure 3-4: AA FIR filter shape

$$\texttt{dout\_i:} \left\langle\!\!\left\langle\ \texttt{0} \cdot -S_6 \cdot \texttt{0} \cdot S_4 \cdot \texttt{0} \cdot -S_2 \cdot \texttt{0} \cdot S_0\ \right\rangle\!\!\right\rangle$$

$$\texttt{dout\_q:} \left\langle\!\!\left\langle\ -S_7 \cdot \texttt{0} \cdot S_5 \cdot \texttt{0} \cdot -S_3 \cdot \texttt{0} \cdot S_1 \cdot \texttt{0}\ \right\rangle\!\!\right\rangle$$

Due to the simplicity of this computation, this module produces valid data in the same clock cycle as its respective input without severely impacting the timing constraints of the system.

## 3.4 Anti-aliasing Filter

The output of the DDC produces an aliased image centered around 500 MHz as shown in Figure 3-3. To remove this component so that it does not affect future stages, it is necessary to create a low-pass anti-aliasing filter.

Because of the large distance between the baseband signal and the aliased component, the filter requirements are not at all very strict; from Figure 3-3, we see that there is somewhere between 300 and 400 MHz of dead space. We chose to use a finite impulse response (*FIR*) filter with 47 taps, generated by the method of least squares. FIR filters are useful because they are architecturally very simple (due to their high degree of symmetry) and are always stable. The filter's shape and frequency response are plotted in Figures 3-4 and 3-5 respectively.

Continuing with our example from the preceding section, we apply this anti-aliasing filter to both the in-phase and quadrature baseband components to achieve the results shown in Figure 3-6.

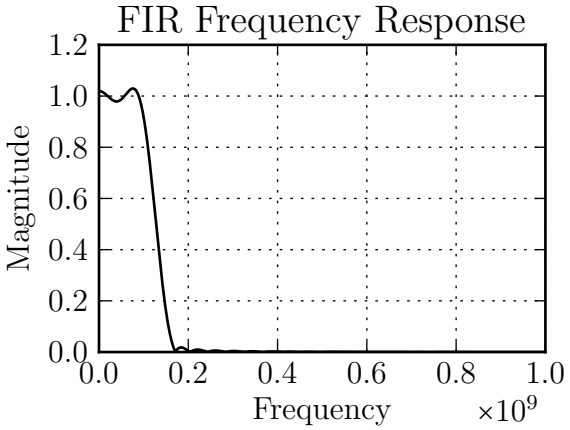


Figure 3-5: AA FIR filter frequency response

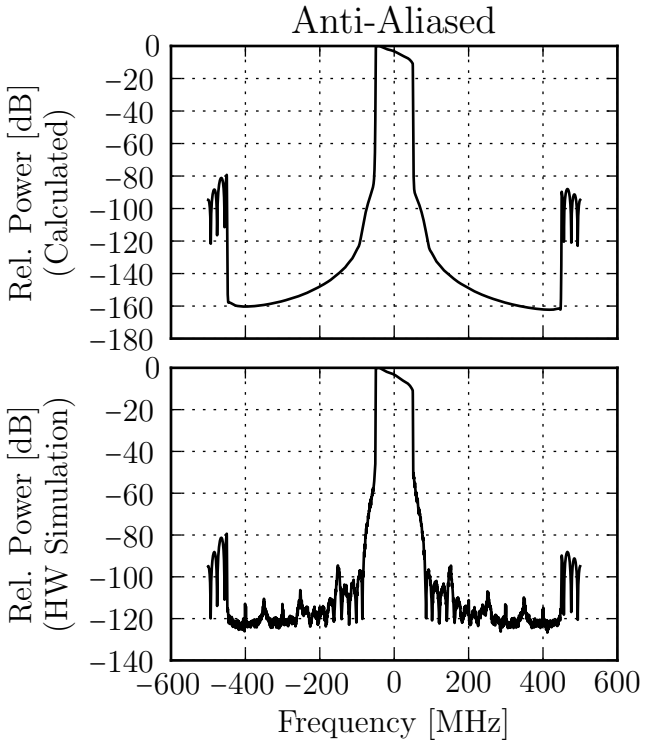


Figure 3-6: Result of applying the anti-aliasing filter to the downconverted signal. Notice that while the overall noise floor of the hardware simulation's result is higher, the overall signal-to-noise ratio is still 80 dB.

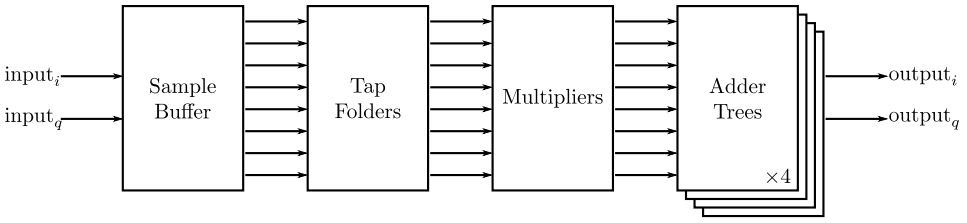


Figure 3-7: **Architecture of the digital anti-aliasing filter**. Input samples are buffered, combined, multiplied, and added together to yield a stream of output samples.

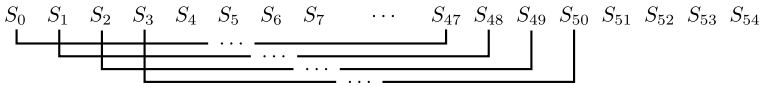


Figure 3-8: **Filter Buffer System**. A 55-sample buffer is kept. 8 "windows" into this buffer are used to produce an octet of output samples. In this figure, the first 4 output windows are shown: the first output sample uses input samples $S_0$ through $S_{47}$, the second output sample uses input samples $S_1$ through $S_{48}$, and so on.

To implement this filter on two octets ($I$ and $Q$), a multi-stage filter was constructed whose overall architecture is described by Figure 3-7. The 7 stages of the system are: buffering, coefficient folding, multiplication, and a four-stage post-multiplication adder tree.

### 3.4.1 Data Capture and Buffering

The first stage is a combination input-buffering and data-loading stage. Given that the filter is 47 taps and the input bus is 8 samples wide, there must be 47 + 8 = 55 samples buffered into the filter at any given point because for each sample in the output bus, a 47-sample window into those 55 samples is used to generate the low-passed result. This design allows us to create an output octet each clock cycle. Figure 3-8 demonstrates how a large shift register keeps these 55 samples buffered in memory.

In the next step, an unregistered bus `stg_a` simply extracts samples from the shift registers. The data structure of this bus is defined as follows:

$$\left[ \begin{array}{cccc} \langle I_0, Q_0 \rangle & \langle I_1, Q_1 \rangle & \cdots & \langle I_{54}, Q_{54} \rangle \end{array} \right] \tag{3.2}$$

### 3.4.2 Tap Folding

The FIR filter is a summation of multiplications, but these multiplications are symmetric around the center tap. For example, for a 7-tap FIR filter as follows:

$$y[n] = C_0x[n] + C_1x[n-1] + C_2x[n-2] + C_3x[n-3] + \tag{3.3}$$
$$C_4x[n-4] + C_5x[n-5] + C_6x[n-6] \tag{3.4}$$

$C_0 = C_6$, $C_1 = C_5$, etc., so the equation can be represented as:

$$y[n] = C_0(x[n] + x[n-6]) + C_1(x[n-1] + x[n-5]) + \tag{3.5}$$
$$C_2(x[n-2] + x[n-4]) + C_3x[n-3] \tag{3.6}$$

There is an important optimization that we can employ in this stage. Because every other sample in the previous stage was 0 and the number of taps of the filter is odd, we can use symmetry to throw out half of the folding additions. This would be analogous to $x[n-1] = x[n-3] = x[n-5] = 0$ in Equation 3.5, which would cause the $C_1(\cdots)$ and $C_3(\cdots)$ taps to drop. Notice how one could not take advantage of this optimization if there were an even number of taps, as each tap would have one non-zero sample and one zero sample.

Therefore, following the extraction of the input shift register into samples, we fold these samples for each of the $(47-1)/2 = 23$ tap pairs for each of the 8 output samples and store the resulting data into the $8 \times 12$ array `stg_b` bus, whose format is described by Equation 3.7.

$$\begin{bmatrix} \langle I_0 + I_{46}, Q_1 + Q_{45} \rangle & \langle I_2 + I_{44}, Q_3 + Q_{43} \rangle & \cdots & \langle I_{22} + I_{24}, Q_{23} \rangle \\ \langle I_2 + I_{46}, Q_1 + Q_{47} \rangle & \langle I_4 + I_{44}, Q_3 + Q_{45} \rangle & \cdots & \langle I_{24}, Q_{23} + Q_{25} \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle I_8 + I_{52}, Q_7 + Q_{53} \rangle & \langle I_{10} + I_{50}, Q_9 + Q_{51} \rangle & \cdots & \langle I_{30}, Q_{29} + Q_{31} \rangle \end{bmatrix} \tag{3.7}$$

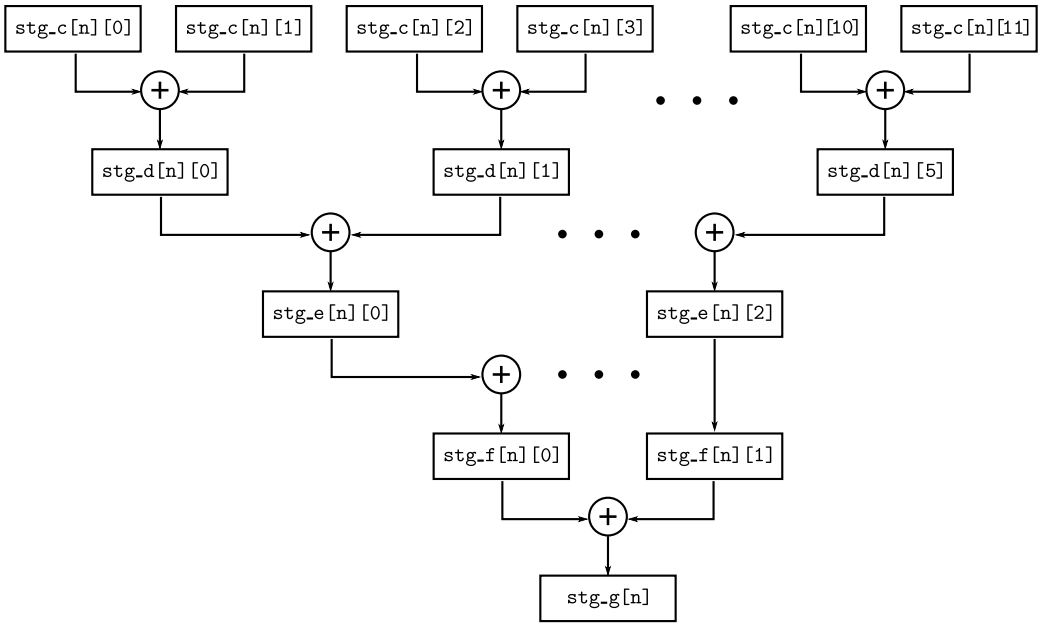


Figure 3-9: **Architecture of the post-multiplication adder tree**. This architecture applies for `n = 0...7` (i.e. all samples on the input bus). Each term is a complex number.

### 3.4.3 Multiplication

In the third stage, the taps in `stg_b` are multiplied by their respective coefficients. Equation 3.8 describes the output of this stage, which is stored as `stg_c`, which again has the dimensions $8 \times 12$.

$$\begin{bmatrix} \langle C_0 \cdot (I_0 + I_{46}), C_1 \cdot (Q_1 + Q_{45}) \rangle & \cdots & \langle C_{22} \cdot (I_{22} + I_{24}), C_{23} \cdot (Q_{23}) \rangle \\ \vdots & \ddots & \vdots \\ \langle C_1 \cdot (I_8 + I_{52}), C_0 \cdot (Q_7 + Q_{53}) \rangle & \cdots & \langle C_{23} \cdot (I_{30}), C_{22} \cdot (Q_{29} + Q_{31}) \rangle \end{bmatrix} \tag{3.8}$$

### 3.4.4 Post-Multiplication Adder Tree

The last four stages of the filter sum the taps in a tree-like fashion. Figure 3-9 illustrates how the components of `stg_c` sum to `stg_d`, which in turn sum to `stg_e` and then `stg_f` and `stg_g`.

## 3.5 Baseband Emulation Stage

The baseband emulation stage is reserved for special modifications applied to the baseband signal. This channel emulator tries to achieve an architecture capable of pluggable transformations, so this stage could easily allow for the addition of additive Gaussian white noise, signal mixing, or other modifications. In this specific implementation, the baseband signal is not modified, however the hardware architecture makes it relatively easy to plug in modules that do so.

## 3.6 Variable Mixer

The final digital processing stage is the mixer, which mixes the baseband signals with a intermediate carrier frequency. The result is sent out via the DAC.

The nominal mix frequency is 120 MHz. That is, given a set of I samples $x_{\mathrm{I}}$ and Q samples $x_{\mathrm{Q}}$, the output signal is:

$$x_{\mathrm{mixed}}[n] = x_{\mathrm{I}}[n]\cos\left(\frac{2\pi f_m}{f_s}n\right) + x_{\mathrm{Q}}[n]\sin\left(\frac{2\pi f_m}{f_s}n\right) \tag{3.9}$$

where $f_m$ is nominally $120 \cdot 10^6$ Hz and $f_s$ continues to be $1 \cdot 10^9$ Hz.

Why nominally? The mixer is the perfect opportunity to inject the effects of $\Delta f_c$ from Equation 2.8, as the entire spectrum can be shifted simply by adjusting $f_m$. $f_m$ becomes $120 \cdot 10^6 + \Delta f_c$.

Figure 3-10 demonstrates the result of mixing the anti-aliased I and Q to 120 MHz.

The digital architecture utilizes a *direct digital synthesizer* (DDS) to generate the carrier wave, multiplies incoming samples by this wave, and sums the results in a fully pipelined operation.

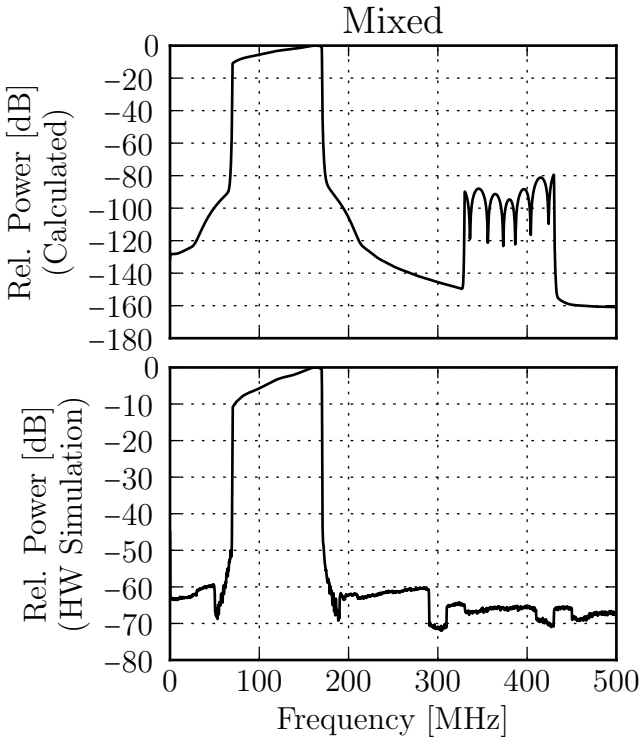


Figure 3-10: Result of mixing anti-aliased baseband signal to 120 MHz. The overall SNR for the hardware implementation is around 60 dB, which, for the target application, is excellent.

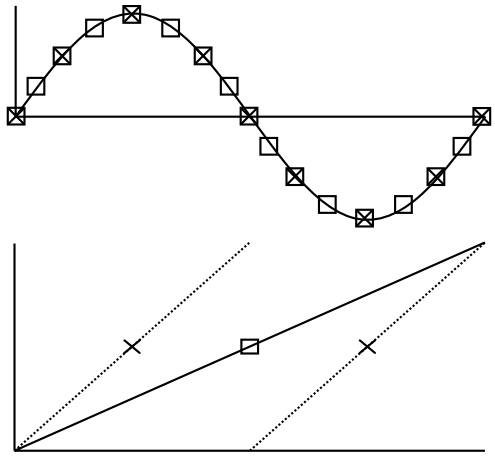

Figure 3-11: **Example of DDS output with varying phase ramps**. As we change the slope of the phase ramp, we select more or fewer elements in a single period of the lookup table. In this example, the dotted line corresponds to samples marked with a × and has twice the slope of the solid line, which corresponds to the samples marked with a □. Clearly, we can see that the resulting × sine wave will have twice the frequency of the □ sine wave.

### 3.6.1 DDS Background Theory

There are many ways to generate sine waves digitally. The simplest method is through the use of a lookup table (LUT). With this method, a period of the sine function (or any periodic function) is stored to memory and a "phase" is applied as an input index into the table. The phase is then incremented and the next value of the function is determined. The input phase accumulates until it overflows back to the start of the periodic function.

The slope of this phase ramp therefore determines the frequency of the output wave. Figure 3-11 illustrates how different phase ramps result in different frequency outputs.

In continuous time, this concept is very simple. For a function $f$ with phase input $\phi$, the output of that function at a given phase is just $f(\phi)$. Furthermore, as we want successive values of $f$, we just update $\phi$ with some $\delta\phi$ continuously. This value determines the frequency. Things become more complicated as we enter the discrete time-domain realm. In this realm, we have a timestep $\tau_s$ instead of being in continuous time as well as a fixed number of positions $N_t$ in a given LUT instead of

having a phase that is all real numbers between 0° and 360°. We now refer to the phase as $\varphi$ and the change in phase at each timestep as $\Delta\varphi$. In a digital system, it makes sense to make $N$ a power of two such that we can always make $\varphi$ simply by creating a register $\lg(N_t)$ bits wide (so that when it overflows, the phase starts at the beginning again).

Now, $\Delta\varphi$ becomes:

$$\Delta\varphi = f_m \cdot \tau_s N_t \tag{3.10}$$

| | |
|---|---|
| $\Delta\varphi$ | increment to phase with each timestep |
| $f_m$ | desired target frequency to generate |
| $\tau_s$ | timestep (inverse of sampling frequency) |
| $N_t$ | number of elements in the LUT |

Unfortunately, unless the quantity on the right-hand side is a whole number, errors accumulate very rapidly as $\varphi \leftarrow \varphi + \Delta\varphi$ with each timestep. As a result, the frequency that is actually generated will be inaccurate. One thing we can do that will sacrifice the purity of the signal (i.e. its precision) in favor of better accuracy is to make $\varphi$ a fixed decimal value and only feed the integral part into the LUT. Thus, with each timestep update, we do not accumulate quite as much error. If we wish to introduce $N_f$ decimal values, with $N_f$ being a power of two, $\Delta\varphi$ becomes:

$$\Delta\varphi = f_m \cdot \tau_s N_t N_f \tag{3.11}$$

Now, we simply take the upper $\lg(N_t)$ bits of $\varphi$ and use that as the index into the LUT.

There are more techniques that one can use to improve the output signal quality in different ways. For example, one natural result of this method is that harmonics are generated. By *dithering* the value of $\Delta\varphi$ with a small random offset on each increment, one can suppress these harmonics at the cost of raising the signal's noise floor.

### 3.6.2 Implementing a DDS

Using a fixed-point decimal notation with the integral and fractional parts both being powers of two makes it easy to separate the components in order to provide an index into the LUT. In this implementation, we set $N_t$ to $2^{10}$ (i.e. 1024 table entries) and $N_f$ to $2^{16}$ (i.e. 16 bits of fractional precision).

The lookup table was implemented as a dual-read-port ROM with 18-bit values. This $1024 \times 18$ design fits neatly into a single Virtex 6 BRAM. On each clock cycle, one of these ports outputs a sine sample and the other outputs the corresponding cosine sample. A total of 8 of these LUTs were instantiated, one for each sample on the input bus.

The nominal mix frequency is $f_m = 120\,\text{MHz}$. We can calculate $\Delta\varphi$ as follows:

$$\Delta\varphi_{\text{actual}} = f_m \cdot \tau_s N_t N_f \tag{3.12}$$

$$= 120 \cdot 10^6 \cdot \frac{1}{1 \cdot 10^9} \cdot 2^{10} \cdot 2^{16} \tag{3.13}$$

$$= 8053063.68 \tag{3.14}$$

$$\Delta\varphi = \lfloor \Delta\varphi_{\text{actual}} \rfloor = 8053063 \tag{3.15}$$

Therefore, with each timestep, we add 8053063 to the existing register storing $\varphi$ and use the top 10 bits of that value as the index into the LUT.

The difference between the actual and approximate phase increment, 0.68, is exceptionally small, and produces a frequency error of only:

$$\text{error} = \left| \frac{\Delta\varphi_{\text{actual}} - \Delta\varphi}{\Delta\varphi_{\text{actual}}} \right| \tag{3.16}$$

$$= 8.44 \cdot 10^{-6}\% \tag{3.17}$$

or around 10 Hz.

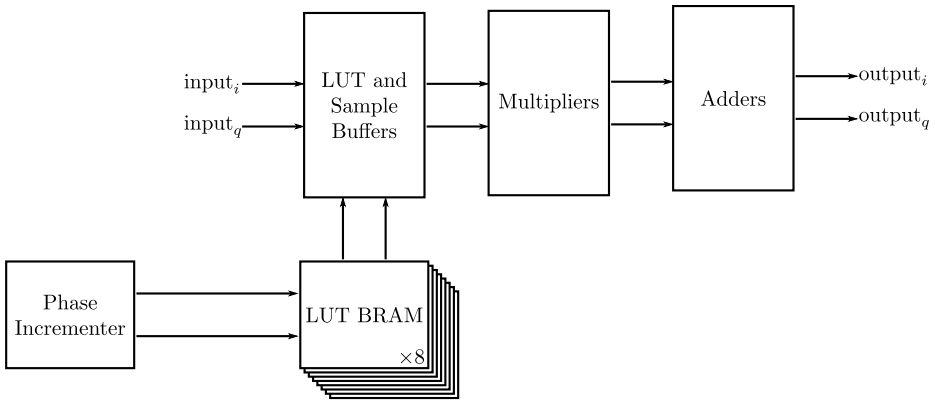


Figure 3-12: **Mixer Architecture**.

### 3.6.3 Mixer Architecture

As illustrated in Figure 3-12, the architecture is a very simple 3-stage pipeline. In the first stage, the set of input samples is registered as well as their corresponding sine and cosine values in the LUT. In the second, the real component of the sample is multiplied by the cosine value and the imaginary component is multiplied by its corresponding sine value. Finally, in the third stage, they are summed.

### 3.6.4 Parallel Lookup Tables

One slight difference between a standard DDS implementation and that of this project is the need for parallel LUTs because there are 8 samples per clock cycle. As a result, we actually need 8 LUTs where we feed offsets of a phase into each one. Namely, if we're starting at the beginning of the table, where $\varphi$ of the first sample is 0, the $\varphi$ of the next sample must be $0 + 8053063$, etc.: Thus, we have an address input bus — a one-dimensional array whose starting addresses are:

$$\left[\begin{array}{cccccccc} 0 & 1\cdot\Delta\varphi & 2\cdot\Delta\varphi & 3\cdot\Delta\varphi & 4\cdot\Delta\varphi & 5\cdot\Delta\varphi & 6\cdot\Delta\varphi & 7\cdot\Delta\varphi \end{array}\right] \tag{3.18}$$

Then, with each timestep, we add $8\cdot\Delta\varphi$ to each of these values.

As mentioned, each LUT has two input ports. This allows simultaneous retrieval of sine and cosine values. The LUT itself is a single period of a cosine function. Therefore, we can retrieve a sine function simply by adding a three-quarter period to

the input phase. For a 1024-entry table, we would add $\frac{3}{4} \cdot 1024 = 768$ to each input phase to get the corresponding sine value.

### 3.6.5 Making the Mixer Variable

Previously, $\Delta\varphi$, was a fixed value, as $f_m$ was set to 120 MHz. However, because the dominant effect of Doppler shifting on a given spectrum is additive frequency change, it makes sense to alter the mixing frequency. From Equation 2.8, we now want to set $f_m = 120 \cdot 10^6 + \Delta f_c$. Because $\Delta\varphi$ is a linear function of the desired mixing frequency, we can represent the term as follows:

$$\Delta\varphi = f_m \cdot \tau_s N_t N_f \tag{3.19}$$

$$= (120 \cdot 10^6 + \Delta f_c) \cdot \tau_s N_t N_f \tag{3.20}$$

$$= 120 \cdot 10^6 \cdot \tau_s N_t N_f + \Delta f_c \cdot \tau N_t N_f \tag{3.21}$$

$$= \Delta\varphi_n + \Delta\varphi_d \tag{3.22}$$

The latter term is the phase increment from the Doppler effect and the former is the nominal increment. Therefore, we generate $\Delta\varphi_d$ and add it along with $\Delta\varphi_n$ to $\varphi$ to generate the next phase input into the LUTs. Figure 3-13 illustrates the additional component of this architecture. This gives us a variable mixer, capable of applying the Doppler effect on an input spectrum.

## 3.7 DAC

The DAC chosen was the Texas Instruments DAC5681Z, which outputs 16-bit samples at 1 GSPS at 1.0 $V_{pp}$. This particular DAC has a SINC output filter that makes production of a signal within the third Nyquist fold impossible. As a consequence, the digital mixer mixes the baseband signal to the intermediate frequency ($IF$) 120 MHz. An external analog mixer then mixes the signal to its final carrier frequency of 1.25 GHz.

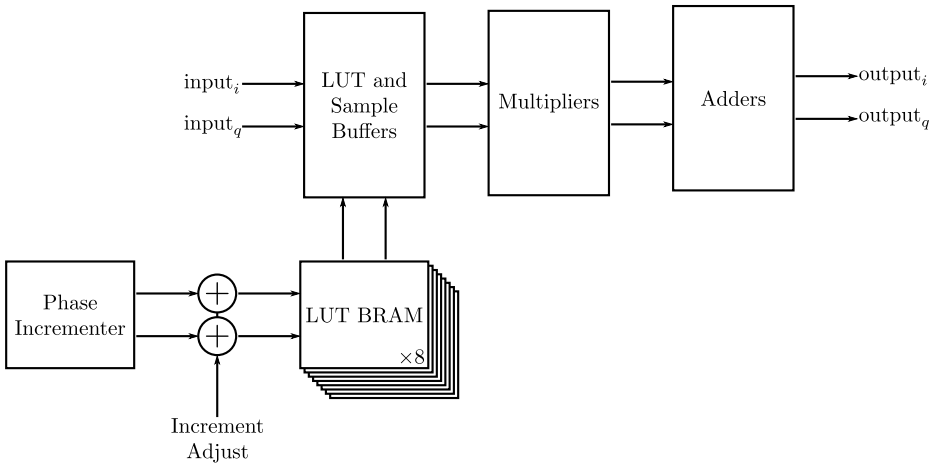


Figure 3-13: **Revised variable mixer architecture**. The addition of the increment adjust input produces a variable mixer.

## 3.8 Analog Mixer

To upconvert from the IF to the carrier frequency, a simple analog mixer is used. For a spectrum centered around $f_m$ to be mixed up to one centered around $f_c$, an analog mixer is fed the local oscillator $(LO)$ $f_c - f_m$ as well as the spectrum at $f_m$. Then, a spectrum is produced at $f_c$ as well as $f_c - 2f_m$. The frequency $f_m$ should be large enough so that the image produced on the opposite side of $f_c - f_m$ is far enough away that it is rejected by the receiving modem's internal passband filter.

## 3.9 Numeric Precision

There is an inherent trade-off between numeric precision and resource usage. All of the calculations involve decimal math. For example, the coefficients in the anti-aliasing filter are all fractions, as are sine and cosine values in the mixer.

Floating-point math is generally not chosen for most digital designs. To minimize unnecessary resource consumption, fixed-point math is generally preferred; thus, it was chosen for this project. Each stage of the signal chain is composed of an integer and decimal component. For example, a 16.8 representation would mean that the integer part was 16 bits wide while the decimal part was 8 bits wide.

One relevant constraint to use when choosing numeric precision is the size of

the hardware multipliers in the FPGA fabric. The Virtex 6 has a large number of DSP48E1 slices, which include a pre-adder, a $25 \times 18$ multiplier, and an ALU. For resource-intensive operations, namely the anti-aliasing filter and the mixer, it would be useful to enforce the constraint that all multiplication inputs are no larger than 25 and 18 bits in width. Based on the layout of the FPGA fabric, in the case of both the AA filter and the mixer, lookup table values are pulled from BRAMs, which dump directly into the 18-bit multiplier inputs.

The input into the AA filter is a 12-bit integer from the ADC. To maintain as much precision as possible while keeping true to the 25-bit limit into the next multiplier stage, the output is 16.9. Table 3.1 lists the fixed-point representations of each stage.

Table 3.1: Numeric format of each signal chain stage's output

| **Stage** | **Output's Representation** |
|---|---|
| ADC Capture | 12.0 |
| DDC | 12.0 |
| Anti-Aliasing | 16.9 |
| Baseband Emulation | 16.9 |
| Mixer | 17.9 |

# Chapter 4

# Implementation of the Channel Emulator

The channel emulator consists of two domains: the hardware domain processes a streaming input signal and the software domain directs the hardware *how* to process that signal. The two communicate via a custom protocol. Careful consideration went into the design of this implementation so that the channel emulator can be extended to emulate environmental effects beyond simple Doppler shifting.

## 4.1 Hardware Considerations

Transforming a signal sampled at 1 GSPS is not possible with a CPU, so the logical choice of signal processing device is a field-programmable gate array (*FPGA*). These devices are best described as blank canvases on which digital circuits of great complexity can be created. FPGAs are perfect for complex signal processing and massively-parallel computations. Creating a digital circuit for an FPGA starts with a behavioral description of that circuit using a hardware description language (*HDL*). In the case of this channel emulator, the language VHDL was chosen. Once the circuit has been fully described, it is *synthesized* and mapped into real logic using a software tool provided by the FPGA vendor. Finally, the result is routed onto the FPGA fabric and a programming file is created. The user can then program the FPGA with

this programming file and run their application.

### 4.1.1 FPGA Evaluation Board

Generally, for the purposes of prototyping, FPGA evaluation kits are used. These kits are full boards that provide useful frameworks for developing applications. In addition to having an FPGA fully routed on the board, they often feature a variety of peripheral hardware, such as USB UARTs, ethernet ports, digital video ports, and an array of buttons and switches for easy configurability.

This project used the ML605 evaluation kit from Xilinx. This board features a Virtex 6 FPGA, DDR3 RAM, high-speed connectivity, and a variety of other features. Xilinx also has excellent documentation for both the features of the FPGA and the components on the evaluation board.

The Virtex 6 family is Xilinx's second-newest generation at the time of writing (and was the newest at the start of the project when hardware decisions were being made). This chip has a variety of useful functionality built into the fabric, and there are few particular features that make it highly desirable for this project. Because the device must operate at very high speeds in a streaming manner, the process of capturing data from the analog interface (described in Section 4.1.2) and sending the transformed signal back to that analog interface requires high-speed hardware support. The Virtex 6 contains a number of high speed serializer-deserializer (*SERDES*) blocks, which convert data between the narrow high-speed bus used to talk to the analog interface and a wide low-speed bus used to process data within the FPGA. This architecture is detailed in Sections 4.2.4 and 4.2.5. Along with these SERDES modules, the Virtex 6 family has clock routing and control features that make processing data across multiple clock domains relatively straightforward. These capabilities are instrumental in producing a real-time streaming signal processing environment.

The last major useful feature of the Virtex 6 layout is the large number of DSP48E1 slices. These are specific function blocks that contain a pre-adder, an 18×25-bit multiplier, and an arithmetic logic unit. In particular, the hardware multipliers are of great importance to this project because they make the process of applying com-

plicated filters to high-speed signals possible. The particular Virtex 6 chip on the ML605 has 768 DSP48E1 slices; the final design of the emulator framework uses a relatively small fraction of them, about 15%. The rest can be used for the individual emulation modules that will come from future development.

### 4.1.2 Analog Interface

The channel emulator's interface to the signal path is analog. Thus, to sample data for processing within the FPGA and then deliver that data to the output, an analog-to-digital converter ($ADC$) and digital-to-analog converter ($DAC$) were necessary. To reduce the development cycle time, a third-party add-on module featuring two ADCs and two DACs was purchased.

4DSP is a company that produces such add-on boards for FPGAs. These boards connect to the FPGA evaluation kit using the FPGA mezzanine connector ($FMC$) pseudo-standard, which provides numerous high-speed connections to the FPGA. 4DSP produces a series of analog boards, but only one fit the suitable requirements: the FMC110. This board contains two ADS5400 ADCs and two DAC5681Z DACs, each operating at 1 GSPS. It also has an on-board clock generator to drive these chips as well as the digital interface to the FPGA.

4DSP provides a reference firmware that can be loaded onto the FPGA to test the features of the FMC110. It sets up the FPGA as a router between the FMC110 and a computer via the evaluation kit's ethernet interface. A Windows program connects to the FPGA, drives a waveform to the DAC and captures a few thousand samples from each ADC. The source code for this firmware is also provided.

The reference firmware is based on a burst architecture: a finite of samples is read into a large buffer, which is then drained more slowly via ethernet to send the data back to the computer. A new design, built from scratch, was necessary in order to meet the real-time streaming requirements of the channel emulator. Components of 4DSP's firmware were used for reference.

This channel emulator only uses one ADC and one DAC. There are plenty of resources on the FPGA available for further development of a second emulation channel

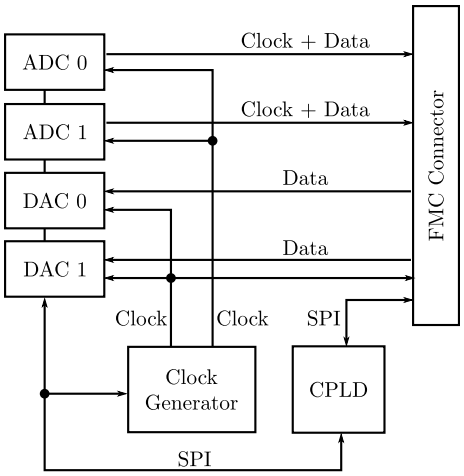


Figure 4-1: **FMC110 architecture**. Relevant components of the FMC110 architecture used by the channel emualtor are shown here.

using the other pair. Figure 4-1 illustrates the architecture of the FMC110, highlighting the parts of the board that are used in the emulator. The ADCs, DACs, and clock generator are configured via their respective serial peripheral interfaces (*SPI*). The high-speed data interfaces on the ADCs and DACs are parallel low-voltage differential signalling (*LVDS*) buses, which consume the majority of the resources of the FMC connector. A configurable programmable logic device (*CPLD*) acts as a router for configuring the devices with SPI interfaces. It also provides the capability to reset certain devices with asynchronous reset lines.

## 4.2 Digital Hardware Architecture

Figure 4-2 is a block diagram of the various hardware components that were implemented on the FPGA. The signal processing architecture described in Chapter 3 is a relatively small component of the larger system operating inside the FPGA.

### 4.2.1 Clocking

It is particularly important to note that there are numerous top-level clocks in the system. Table 4.1 describes their origin and purposes. Note that clocks with signals {`p,n`} are differential and are used to drive `MMCM` blocks on the FPGA to generate

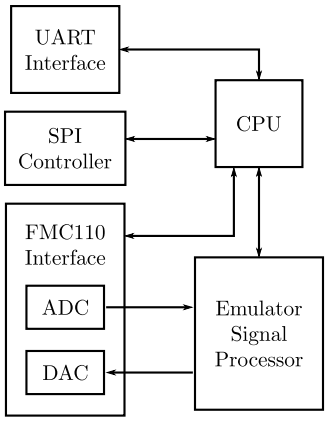


Figure 4-2: **Digital hardware architecture**. Not shown are lesser components such as the PWM fan speed controller and debugging modules, which were later removed in the final design.

other clocks.

### 4.2.2 Picoblaze Microprocessor

After the FPGA starts up, the system must perform a series of configuration steps to initialize all of the on-board hardware. After the system has been configured, it can continue to receive additional configuration directives. It is common in most FPGA designs to include a small microprocessor on which to allocate portions of the design that are state-based, such as reading in a command string.

The Picoblaze is a very small soft-core microprocessor developed by Xilinx engineer Ken Chapman. While minimalistic, it provides the perfect feature set for configuration of various modules on the device.

Firmware for the microprocessor is written in assembly, assembled, and the resulting machine code is loaded into block RAMs on the FPGA. The firmware implementation is discussed in more detail in Section 4.3.

### 4.2.3 SPI Controller

The FMC110 board is configured via a SPI bus. SPI commands sent to the board are interpreted by the on-board CPLD, which directs communication to one of the configurable chips on the device.

Table 4.1: System Clocks

| Clock Name | Freq. | Source | Main Uses |
|---|---|---|---|
| sysclk_{p,n} | 200 MHz | FPGA Eval Board | Generation of other board clocks |
| clk200 | 200 MHz | sysclk_{p,n} | ADC IODELAY control |
| clk125 | 125 MHz | sysclk_{p,n} | virtually all processing and control blocks |
| clk50 | 50 MHz | sysclk_{p,n} | fan PWM clock generator |
| pwmclk | 25 MHz | clk50 | fan PWM clock |
| adc0_clka_{p,n} | 500 MHz | FMC110 | input data clock from ADC0 |
| adc0_dout_dclk | 125 MHz | adc0_clka_{p,n} | ADC0→FPGA output data samples |
| clk_to_fpga_{p,n} | 500 MHz | FMC110 | DAC input data |
| dac0_din_dclk | 125 MHz | clk_to_fpga_{p,n} | FPGA→DAC input |

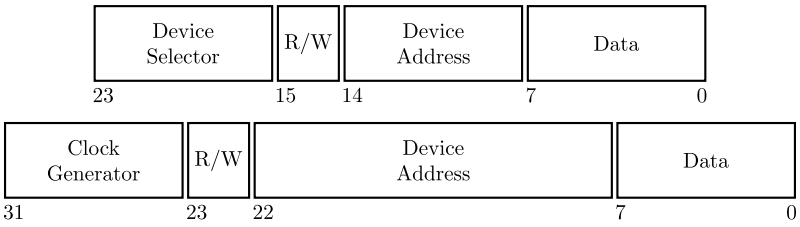


Figure 4-3: **FMC110 SPI packet structure**. SPI packets begin with a prefix that indicates which device the controller is intending to write to or read from. All devices except for the clock generator use an 8-bit address as shown in the upper diagram. The clock generator uses a 16-bit address as shown in the lower diagram.

To facilitate SPI communication with the FMC110, a SPI controller was built. The microcontroller firmware provides a set of subroutines that send data to output ports that are connected to the SPI controller, which, in turn, processes the SPI instruction. Figure 4-3 illustrates the structure of a SPI packet. The ADCs, DACs, and CPLD all have an 8-bit address space and the clock generator has a 16-bit address space; the SPI controller handles both cases.

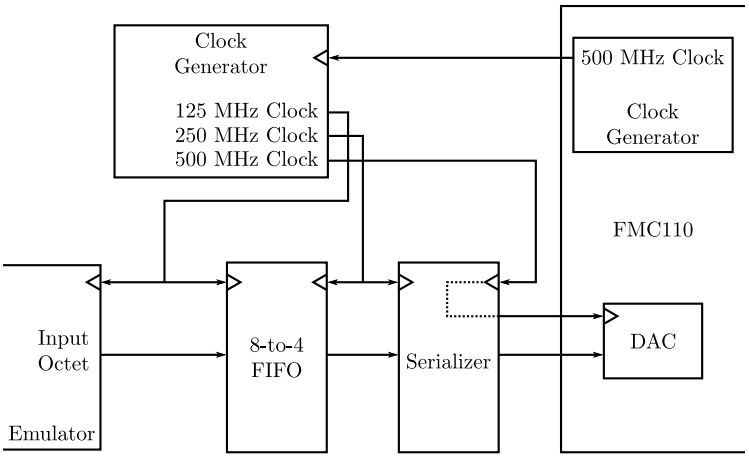


Figure 4-4: **FMC110 DAC controller architecture**. The DAC controller uses an incoming clock from the FPGA to produce its own set of clocks for driving data flow. The serializer sends a data stream to the DAC, as well as a 1 GHz clock.

### 4.2.4 DAC Controller

Fundamentally, to drive the DAC, 16 bits of data must be fed in parallel output format at a speed of 1 GHz. Given that the FPGA itself operates at a speed of 125 MHz, achieving this design constraint is done by passing the data through several clock domains. Figure 4-4 describes stages of the DAC controller.

#### Clock Generation

The FMC110 provides a 500 MHz signal to the FPGA that is synchronized to the data-loading clock it internally routes to the DACs. This clock is divided into two new clocks: one at 250 MHz and one at 125 MHz named `fmc_clk_d4` and `fmc_clk_d8` respectively. The clock line `fmc_clk_d8` is provided as an output from the module and incoming data must be aligned it.

#### 8-to-4 FIFO

Data entering the module aligned to `fmc_clk_d8` is sent into a FIFO whose read clock is `fmc_clk_d4` and whose read port size is a quartet of samples. Thus, if the sample input at a given time was $\llcorner\!\langle\!\langle A \cdot B \cdot C \cdot D \cdot E \cdot F \cdot G \cdot H \rangle\!\rangle\!\urcorner$, then the outputs from the FIFO would be: $\llcorner\!\langle\!\langle E \cdot F \cdot G \cdot H \rangle\!\rangle \rightarrow \langle\!\langle A \cdot B \cdot C \cdot D \rangle\!\rangle\!\urcorner$.

#### Reordering and 4-to-1 Serialization

With data now formatted as quartets at 250 MHz, the data bits are reordered for input into a SERDES serializer, which produces individual samples aligned to the 500 MHz input clock at double data rate. These samples are sent out to the DAC via the FMC110 interface. Additionally, the serializer actually generates a 1 GHz clock (by alternating `1` and `0` bits at 500 MHz DDR) and sends that to the DAC as well.

### 4.2.5 ADC Controller

The ADC interface is somewhat the "opposite" of the DAC's: serial samples at the hardware interface are converted to blocks of parallel samples synchronized to the FPGA's system clock. However, it is somewhat more complicated in that its incoming data lines must be synchronized to the incoming clock using a delay-locked loop (this initialization phase is discussed in Section 4.3.4). Figure 4-5 illustrates the processing stages of the ADC controller.

The FMC110 delivers data on a supplied 500 MHz input clock. An MMCM produces a 125 MHz clock. The incoming data is sent through a SERDES de-serializer whose output is an octet of samples aligned to the divided clock. The output is then sent through a FIFO in order to align the data to the FPGA system clock.

### 4.2.6 FIFO Dumper

To test and verify the operation of the ADC, a FIFO dump system was developed. When the CPU triggers the dumper to start, the output of one of the ADCs is redirected from going into the channel emulator's signal chain to a very large FIFO (it was configured to take up almost all of the remaining BRAM on the FPGA). When the FIFO is full, the dumper takes control of the UART and sends its full contents out via the UART. A software script reads this data and unpacks it back to its 12-bits-per-sample representation.

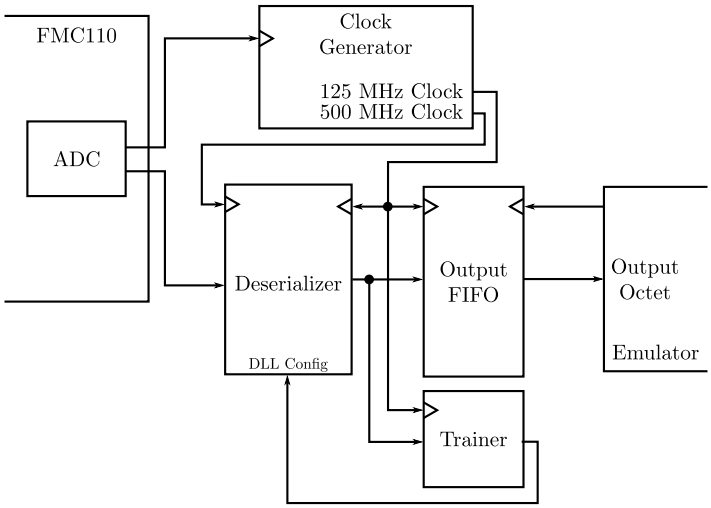


Figure 4-5: **FMC110 ADC controller architecture**. The ADC controller features a DLL trainer that configures the de-serializer; its functionality is described in Section 4.3.4.

## 4.3 Configuring FMC110 Hardware

All of the devices on the FMC110 must be properly initialized. The overall startup process is as follows:

1. Configure the CPLD to set board-level settings.
2. Configure the clock generator to provide clocks to the ADCs, DACs, and FPGA.
3. Configure the DACs and have them perform a "training" phase so that they can align their clock delay appropriately
4. Configure the ADCs and perform a training phase on the FPGA to align clock delays.

This startup process is driven by the CPU. Nearly 80 registers need to be set, making the startup code fairly involved. The startup takes around 10 seconds to complete.

### 4.3.1 Channel Emulator Configuration Interface

The Picoblaze Assembler has no support for macros, so a small interpreted language was designed to produce assembly code from a set of keywords. This program was named `cci` (*channel emulator configuration interface*). For example, if we wish to configure ADC0's register `0x00` to have value `0xAA`, then we would write as an input to `cci`:

```
fmcw adc0 00 AA
```

`cci` would then produce an assembly file containing the code:

```
load s0, 00
load s1, AA
call send_adc0
```

Appendix A describes the full syntax of the language and details how `cci` can be used to both generate assembly code and to perform on-the-fly configuration.

The channel emulator makes use of `cci` to generate the initialization routine which brings up all of the chips.

### 4.3.2 Configuring FMC110's CPLD and Clock Generator

After the CPU has started, it jumps into the FMC110 initialization routine, which configures the CPLD to reset the clock generator and use an internal clock source. It also resets the DACs. Next, it configures the clock generator, which provides stable, phase-corrected 500 MHz clocks to each ADC and DAC, as well as one to the FPGA.

The routine then waits for the PLL on the clock generator to lock before proceeding to configure the DAC.

### 4.3.3 Configuring DAC0

The DAC is configured to provide a two's complement output and the FPGA is then configured to provide a stable data clock to the DAC. The DAC's DLL is then reset

and the CPU waits until the DAC reports that its DLL has been locked. This process guarantees that the DAC has synchronized to the data clock that the FPGA provides along with the input data stream.

Once the DLL has locked, the FPGA provides a checkerboard pattern of data: a stream of `0xAAAA`, `0x5555`, `0xAAAA`, etc. It then reads the DAC error register to confirm that no errors in the checkerboard pattern were received. This stage ensures that the data going into the DAC is correctly aligned to the data clock.

Following the training period, the CPU reports if there were any errors in testing by reading the error register of the DAC. It then enables the DAC for standard operation (streaming data output).

### 4.3.4 Configuring ADC0

Finally, the CPU configures the ADC for operation. Before the input data stream can be considered valid, the ADC controller must undergo a training phase, which ensures that each incoming data bit is aligned to the incoming clock. Because the data is being sent from the ADC extremely quickly, the length of the traces can affect propagation time. It is possible for the clock to become skewed from the data, which would cause the data to sometimes appear invalid. To overcome this, a delay-locked loop is available for every incoming bit in the sample stream. In the training phase, the ADC is configured to produce a pseudo-random bit sequence (*PRBS*) with the function $x^7 + x^6 + 1$. The training system calculates the same PRBS on the FPGA and compares the result to what it receives from the ADC. When it receives incorrect data, it adjusts the SERDES delay timing and tries again.

Finally, the CPU confirms the validity of the ADC input by configuring it to send a checkerboard stream (`0xAAA`, `0x555`, `0xAAA`...) and tests for received values that deviate for the pattern. After a few seconds of testing, it reports back the number of times it discovered an error.

The CPU sets the ADC to run in normal mode and at that point, initialization is complete.

## 4.4 Configuring the Signal Processing Chain

There are two main signal processing chain settings. The first is the direction of the baseband down-converter. By default, the converter runs in counter-clockwise mode, meaning that it multiplies the set of incoming samples by the rotation $\lfloor 1, j, -1, -j \rceil$. However, if the incoming spectrum is inverted, it can be fixed by changing to a clockwise rotation: $\lfloor 1, -j, -1, j \rceil$. Using `cci`, one can change this direction on the fly.

The other configuration option is the digital gain setting. When the signal processing chain is performing mathematical operations on the input signal, it preserves as much resolution as possible. Then, at the final output, 16 bits of the number must be selected. The digital gain setting determines how far to shift the final 26-bit value from its most significant bit. By default, a setting of 6 is used, meaning that bits 19 through 4 of the output will be sent to the DAC. A utility allows the user to get and change the digital gain.

There is one more useful gain to configure: the ADC's analog gain. This is set and read from a register using SPI. The gain adjustment utility can modify this value as well.

## 4.5 Applying Doppler Shifting

The Doppler shifting effect is applied by adjusting the mix frequency ($f_m$ in Equation 3.9). To do this, the CPU configuration bus is used to apply an offset to the phase slope increment of the mixer.

As discussed in Section 3.6, the phase slope is fed into a lookup table which returns a value of a sine or cosine at the appropriate phase point. By adjusting the amount the phase slope increments with each timestep, one can change the frequency of the generated sine wave on the fly. The `mix` module is given a 24-bit `incr_adj` value that changes the phase slope increment on the next clock cycle.

Since data transfered between the computer and the CPU on the FPGA is 8-bits in width, a frequency shift is represented by configuring 3 *times* 8-bit registers on the

FPGA and then setting a fourth register to `0x01` to indicate that the next frequency shift is valid. The total time to issue a frequency shift directive is largely limited by the baudrate, or approximately $\frac{4 \cdot 10}{460800} = 0.0000861\,\text{sec} = 86\,\mu\text{s}$, which is also around 11520 frequency shifts per second.

# Chapter 5

# `dopgen` Software Package

A software package named `dopgen` was written to smoothly apply frequency shifting directives to the channel emulator by mathematically transforming sets of transmitter and receiver coordinates into frequency shifts. It supports two main modes of operation: processing a pre-recorded set of coordinates and receiving data live from a flight simulator.

The first mode is exceptionally useful when using captured data from physical scenarios. However, because most captured data is taken at relatively slow intervals (usually once or twice per second), `dopgen` is capable of interpolating between data points, sending frequency shift commands at faster intervals to the channel emulator to allow for smoother operation.

By performing frequency shift computations in software and then sending directives to the FPGA board, the process of adding additional features (such as generating frequency shifts from more complicated antenna patterns) is greatly simplified.

## 5.1 System Architecture

Figure 5-1 illustrates the architecture of `dopgen`. The main thread begins by parsing command line arguments, determining whether to run in replay mode or live mode. It then creates an instance of the main `Processor` object, which takes coordinates, calculates the frequency shift, and sends the appropriate command to the hardware.

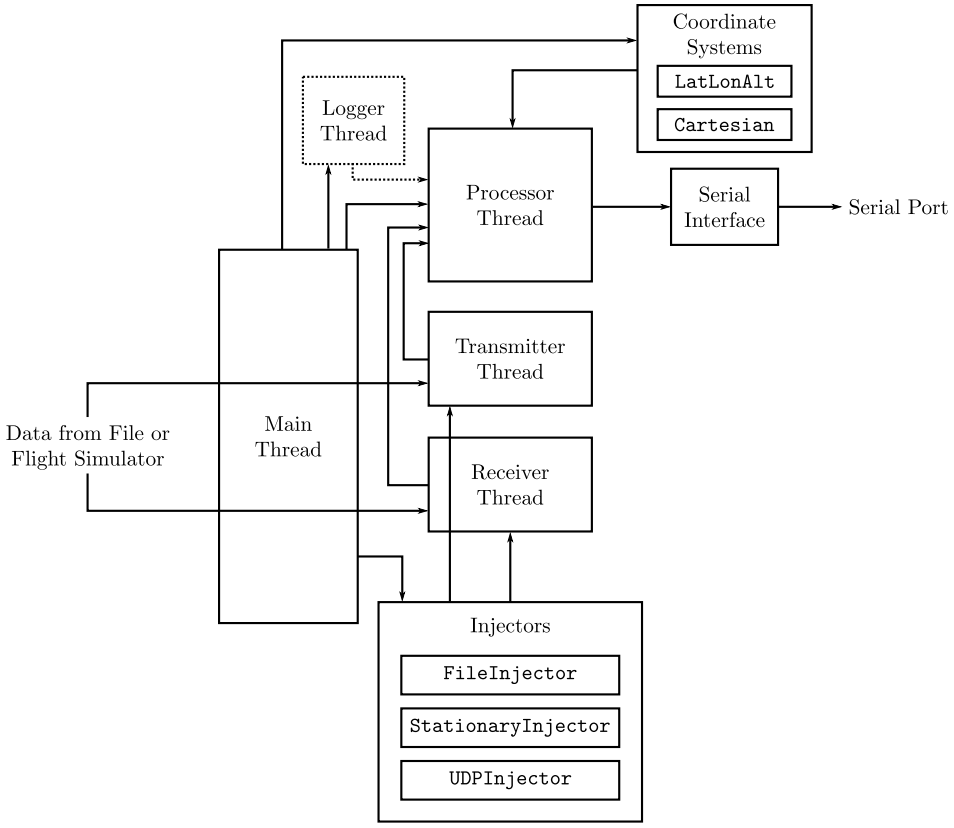


Figure 5-1: `dopgen` **Software architecture**. `dopgen`'s architecture is heavily threaded. The processor thread does most of the heavy lifting — computing frequency shift directives from transmitter and receiver coordinates and velocities and then communicating those directives via the serial interface. The transmitter and receiver threads periodically inject such coordinates into the processor.

To supply the processor with antenna coordinates, the main thread instantiates two `Injector` instances, one for the transmitter antenna and the other for the receiver antenna. It then configures each appropriately so that they may both feed the processor coordinates and indicate when they have finished.

### 5.1.1 Injectors

The `Injector` class provides a method to asynchronously provide the main processing unit with an update on the coordinates of a given antenna. Each injector instance runs in its own thread. Upon initialization, it is provided with two functions pointers from the `Processor` object. The injector calls the first when it wishes to update the coordinates on the processor and the second to indicate that it has finished supplying all of its coordinates.

Three types of injectors were built.

#### `FileInjector`

The `FileInjector` takes a set of predefined samples and injects them into the channel emulator according to timestamps provided with each sample.

It is initialized with the set of samples in the following format:

```
timestamp : [latitude, longitude, altitude]
```

Because the data is provided in dictionary form, with the timestamp being the key and the coordinates the value, it is not necessary to provide timestamps that are always separated by the same interval. Furthermore, for a given system with two `FileInjector` instances (one for the transmitter and one for the receiver), the timestamps can be completely independent of one another; the system only cares about the relative time between each timestamp.

This injector simply injects each coordinate and then waits for the time between the two successive timestamps. When it has injected all of the coordinates, it calls the `donefn()`.

`UDPInjector`

This injector is supplied with a hostname and port to bind to upon initialization. It binds to this UDP port and upon receiving a data packet, it extracts the antenna coordinates and injects them into the processor.

The format of the packet is a set of packed 32-bit floats that represent data such as the antenna's latitude, longitude, and feet above mean sea level. Since it is a streaming server, this injector must be stopped externally, by invoking its `stop()` method.

`StationaryInjector`

This is the simplest injector. It is simply supplied with an initial coordinate, which it injects into the processor and then stops itself. It is meant to be used alongside `UDPInjector` in cases when there is only one instance of a live data source available.

### 5.1.2 Processor

The `Processor` class sets up an object that converts the coordinates of a transmitter and receiver into a frequency shift and then sends the appropriate shift command to the hardware.

As discussed in Section 5.1.1, this class provides two function pointers to each injector thread: `injfn` provides a path for the injector to supply the processor with new coordinates and `donefn` allows the injector to inform the processor that it has finished supplying coordinates.

The processor thread performs a complex dance to get all the timing right, but the essence of its operation is that it waits for either one of the injectors to supply a new coordinate, updates the system model, and generates a new frequency shift directive. If interpolation is enabled, it will perform this update if the interpolation timestep has passed since the last update or if an injector fires a new coordinate at the processor.

The actual process of updating the environment model begins with the establish-

ment of a coordinate system. To compensate for the fact that `dopgen` offers data input from multiple different sources, a set of two `CoordSys` coordinate system classes were created, one for Cartesian inputs and another for geographic inputs. These classes provide the appropriate mathematics for computing relative position, velocity, radial velocity, etc. Section 5.1.3 provides a detailed explanation of each calculation.

After receiving an event (such as new coordinates from an injector or interpolation timestep), the processor updates the coordinates of each antenna. If there are new coordinates waiting from one of the injectors, the processor simply updates its model to have those coordinates. If there are no new coordinates available and interpolation is enabled, it instead sets coordinates by looking at the previous two sets of received coordinates, calculating the velocity and interpolating a result.

Following the establishments of the transmitter and receiver coordinates, the processor calculates the relative position and relative velocity vectors of the system. It then uses those to calculate the radial velocity, which was defined by Equation 2.21. It also applies a multiplicative scaling factor if the user supplied one.

From the radial velocity, the frequency shift and the associated directive is computed and sent via the serial interface. Finally, the processor checks to see if both antennas have stopped moving. If that's the case, the thread exits; otherwise, it waits for another coordinate or interpolation event.

### 5.1.3 Coordinate Systems

In order to easily support multiple input formats, the actual mathematics process was abstracted away from the main `Processor` class in the form of coordinate system transformations.

Namely, when the user runs `dopgen` with various input arguments, the program selects the correct coordinate system to represent the environment based on the type of input the user gives. The coordinate system can be overridden with a configuration switch. Each system provides methods to perform the following computations:

1. **Relative position**, given two position vectors
2. **Velocity**, given two position vectors and a time differential
3. **Relative velocity**, given two velocity vectors
4. **Radial velocity**, $v_r$ as calculated by Equation 2.21
5. **Shift frequency**, $\Delta f$ according to Equation 2.7
6. **Increment adjust**, the configuration directive sent to the hardware
7. **Interpolation**, a simple linear interpolation that produces a new set of coordinates given a previous position vector, a velocity vector, and the amount of time elapsed: $\vec{u}_{\text{new}} = \vec{u} + \Delta t \cdot \vec{v}$

The base class computes all of these functions in the Cartesian coordinate system: i.e. the inputs are some absolute vectors in $xyz$-space. The geographic coordinate system system first transforms the input vectors into a Cartesian representation using Equations 5.1–5.3 and then acts identically to the Cartesian coordinate system thereafter.

$$\vec{u}_x = \left(r_{\text{earth}} + \vec{u}_\xi\right) \cdot \sin\left(\vec{u}_\phi\right) \cdot \cos\left(\vec{u}_\lambda\right) \tag{5.1}$$

$$\vec{u}_y = \left(r_{\text{earth}} + \vec{u}_\xi\right) \cdot \sin\left(\vec{u}_\phi\right) \cdot \sin\left(\vec{u}_\lambda\right) \tag{5.2}$$

$$\vec{u}_z = \left(r_{\text{earth}} + \vec{u}_\xi\right) \cdot \cos\left(\vec{u}_\phi\right) \tag{5.3}$$

| | |
|---|---|
| $\vec{u}_x$ | Cartesian $x$ component |
| $\vec{u}_y$ | Cartesian $y$ component |
| $\vec{u}_z$ | Cartesian $z$ component |
| $\vec{u}_\xi$ | Altitude above mean sea level |
| $r_{\text{earth}}$ | Average Earth radius |
| $\vec{u}_\phi$ | Latitude |
| $\vec{u}_\lambda$ | Longitude |

### 5.1.4 Serial Interface

The `SerialInterface` class generates a connection to the hardware via a USB serial port. Upon instantiating an object, the programmer specifies the device name and baud rate. The class provides methods to initialize the hardware prompt and send frequency shift directives through the `cci` protocol.

### 5.1.5 Logger

To facilitate data visualization, `dopgen` will log data with each new frequency shift request. Each log entry contains:

1. a timestamp
2. transmitter and receiver positions
3. transmitter and receiver velocities
4. relative position and velocity
5. radial velocity, frequency,
6. (optionally) received modem data

To append additional information about the currently running emulation, `dopgen` can optionally connect to a TCP server that serves data from the testing modem. One such server was written to communicate with the Newtech AZ-410 modem's web

interface to receive information about the modem's error rate, offset frequency, and other statistics.

## 5.2 Using Data from Pre-Recorded Scenario

`dopgen` can read antenna position data from a pre-recorded scenario file in one of two formats: the asynchronous EEL format and the synchronous CSV format.

### 5.2.1 The EEL Pseudo-Standard

The Emulation Event Log format was created by the PROTEAN research group at the US Naval Research Laboratory. Each line of the file contains a timestamp, a node number, and the position of that node. `dopgen` interprets a subset of the format to capture position data of a transmitter and receiver antenna. A typical line of the file looks as follows:

```
0.000000 nem:1 location gps 16.591267,15.445880,1000.000000
```

Table 5.1 describes the fields.

Table 5.1: Format of the EEL file

| **Component** | **Meaning** |
|---|---|
| `0.000000` | Timestamp, in seconds |
| `nem:1` | Node number |
| `location gps` | Next field will be GPS coordinates |
| `16.591267` | Latitude |
| `15.445880` | Longitude |
| `1000.000000` | Altitude above mean sea level in meters |

`dopgen` uses node `1` as the transmitter and node `2` as the receiver. This file format is asynchronous because a given line only specifies one node, meaning that the transmitter and receiver antennas can be updated on different timestamps. That is, the two antennas operate independently.

### 5.2.2 CSV File Format

In order to support a more simple processing scheme, `dopgen` also interprets CSV files. These files are *synchronous* in that in a single line, there is a timestamp and the coordinates of both the transmitter and the receiver. The transmitter and receiver must therefore be updated on the same timestamps.

A typical line of the CSV file looks as follows:

```
0.0,100.0,200.0,300.0,110.0,210.0,310.0
```

Table 5.2 describes the fields.

Table 5.2: Format of the CSV file

| **Component** | **Meaning** |
|---|---|
| `0.0` | Timestamp, in seconds |
| `100.0` | Transmitter $x$ |
| `200.0` | Transmitter $y$ |
| `300.0` | Transmitter $z$ |
| `110.0` | Receiver $x$ |
| `210.0` | Receiver $y$ |
| `310.0` | Receiver $z$ |

By default, the CSV file is interpreted with Cartesian coordinates, but it can also supply geographic coordinates.

## 5.3 Injecting Data from Flight Simulators

`dopgen` interfaces with the commercial flight simulator X-Plane to support real-time application of the Doppler effect to a data stream. It can be configured to receive data from one or two instances of X-Plane depending on whether the user wishes to have a stationary antenna or two moving antennas. X-Plane provides a variety of advanced functionality that allows users to record and replay advanced flight paths and fly multiple planes in the same environment.

`dopgen` listens on UDP ports for transmitter and receiver coordinates. X-Plane is then configured to transmit plane position data to one of `dopgen`'s listening ports. This data is a set of packed 32-bit floating point integers that include the current latitude, longitude, and altitude of the plane. With each received packet, `dopgen` extracts these coordinates and injects them into the processor.

This mode is especially useful for generating and testing realistic flight paths as well as demonstrating the use of the channel emulator in real-world scenarios.

# Chapter 6

# Evaluating the Emulator's Performance Capabilities

The first step in verifying the functionality of the channel emulator was to get an idea of its operation in the target channel bandwidth. The original design goal stated that when no Doppler shifting was applied, the channel emulator should be able to transparently pass a 100 MHz signal centered around 1.25 GHz. Therefore, if we were to pass a signal with some set of spectral characteristics between 1.2 GHz and 1.3 GHz through the channel emulator, we would expect to see an almost identical spectrum within that region on the output of the emulator. To test the emulator's actual bandwidth limit, two methods were used. In the first, a low-bandwidth modem's output was swept across a range of frequencies and its performance at each point was measured. In the second, a high-bandwidth modem was fed into the channel emulator with successively larger symbol rates until the modem started reporting data errors.

The experimental results matched well with one another. Both experiments indicated that the emulator's actual bandwidth limit was around 122 MHz to 125 MHz.

## 6.1 Bandwidth Limit Evaluation using a Newtec AZ-410 Modem

The Newtec AZ-410 modem is capable of sending and receiving a modulated PRBS signal in order to evaluate communication errors. This modem is primarily used as a satellite modem and employs the DVB-S2 standard. One interesting property of DVB-S2 is that its operating point on its $E_b/n_0$ curve is such that the error rate is either very high or very low. As a consequence, a signal will almost always appear in sync and receiving cleanly or not at all. This fact is useful in testing the performance capabilities of the channel as well as in evaluating DVB-S2's resistance to Doppler shifting.

A series of tests were conducted with the following settings to determine some preliminary characteristics of the channel emulator's bandwidth capabilities. The testing modem was configured to use a QPSK $\frac{9}{10}$ code with signal bandwidth of around 18 MHz. In order to get an idea of the spectrum capabilities of the channel emulator, the modulation and demodulation center frequencies of the modem were adjusted, sweeping across a broad range of frequencies. That is, the channel emulator's center frequency was kept constant at 1250 MHz and the modem was configured to operate within different portions of the channel's frequency range.

Specifically, we expected that for modulation/demodulation center frequencies from $1250 - (100/2) + (18/2) = 1208$ MHz to $1250 + (100/2) - (18/2) = 1292$ MHz, the emulator should cleanly and transparently forward the input spectrum. Additional testing was done for center frequencies outside this target spectrum.

Figure 6-1 illustrates the test configuration. Modem output is sent into the channel emulator, which then directed back into the input of the modem and is also sent into an Agilent Vector Signal Analyzer, which calculates signal performance characteristics and stores baseband samples.

The error vector magnitude ($EVM$) is a measure of the quality of a received signal's constellation. Values closer to 100% indicate higher deviation of received samples from the ideal constellation points. Thus, the EVM is a good measure of overall sig-

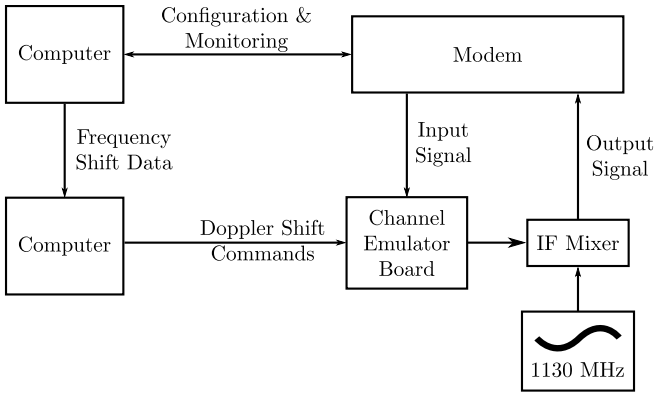


Figure 6-1: **Channel emulator test setup**. This setup was used to test the functionality of the channel emulator. A computer connected to the modem supplied modulation and demodulation parameters and also retrieved frequency shift data, which it forwarded to the other computer, which ran `dopgen` to control the channel emulator and log statistics.

nal quality. To obtain a preliminary verification that the channel emulator operates on the target 100 MHz bandwidth, the AZ-410 was configured to operate at several different center frequencies along the 1.2 GHz to 1.3 GHz range. Figure 6-2 plots the EVM as a function of the modulation/demodulation center frequency. Within the target 100 MHz bandwidth, the EVM is below 25%, which is very good. In the center of the channel, at 1250 MHz, it is around 14%, which is nearly identical to the EVM of the case where the modem's output is looped directly back into the input, bypassing the channel emulator entirely.

Figures 6-3 through 6-12 show the spectra received by the modem given some modulator/demodulator center frequency. At the outer bounds (e.g. when $f_c$ = 1180 MHz, there is significant aliasing, causing the modem to fail to receive data. For these plots, the spectrum gain was set as high as possible to maximize the SNR and no Doppler shift was applied.

The figures show that the channel emulator can accurately emulate the target bandwidth of 100 MHz. That is, from 1200 MHz to 1300 MHz, the emulator can recreate the input spectrum without any serious degradation in signal quality. Furthermore, the channel emulator does slightly better than this: the emulator appears to be able to accurately operate between around 1190 MHz to 1315 MHz, yielding a

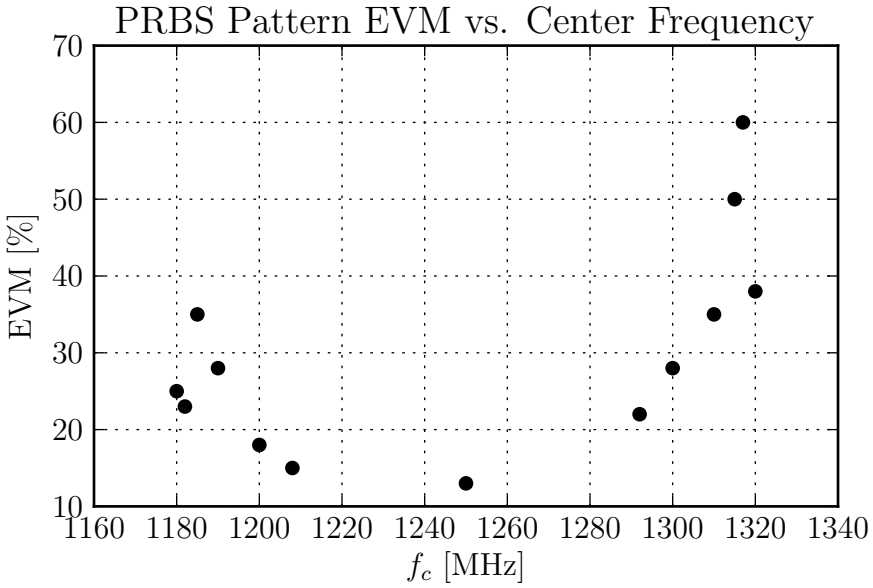


Figure 6-2: **EVM as a function of** $f_c$. At the outer ranges, the EVM increases sharply.

bandwidth of 125 MHz and a bandwidth margin of around 25 MHz.

## 6.2 Testing with High-Bandwidth Modem

The channel emulator was also tested with a second modem, which also used the DVB-S2 standard. This modem was configured with a QPSK $\frac{1}{2}$ code and was transmitting scrambled null frames (effectively looking like random data). Figures 6-13 through 6-16 show the resulting spectra from testing the channel emulator with successively larger input bandwidths. The modem was commanded with a target sample rate and because it uses a 35% roll-off factor, the resulting bandwidth of the signal is simply $1.35 \cdot S_r$ where $S_r$ is the sample rate. At 74 MSPS, the bandwidth is 99.9 MHz.

The test setup looks identical to that used for the AZ-410 modem, as illustrated in Figure 6-1. In this case, samples from the Modem's ADC were captured directly to generate the plots below. For these tests, the low-pass analog filter on the channel emulator's DAC was removed.

There is one slight issue with this modem. Its sampling frequency is 1.2 GSPS, so a spectrum of 100 MHz centered around 1.25 GHz would result in the lower end of the

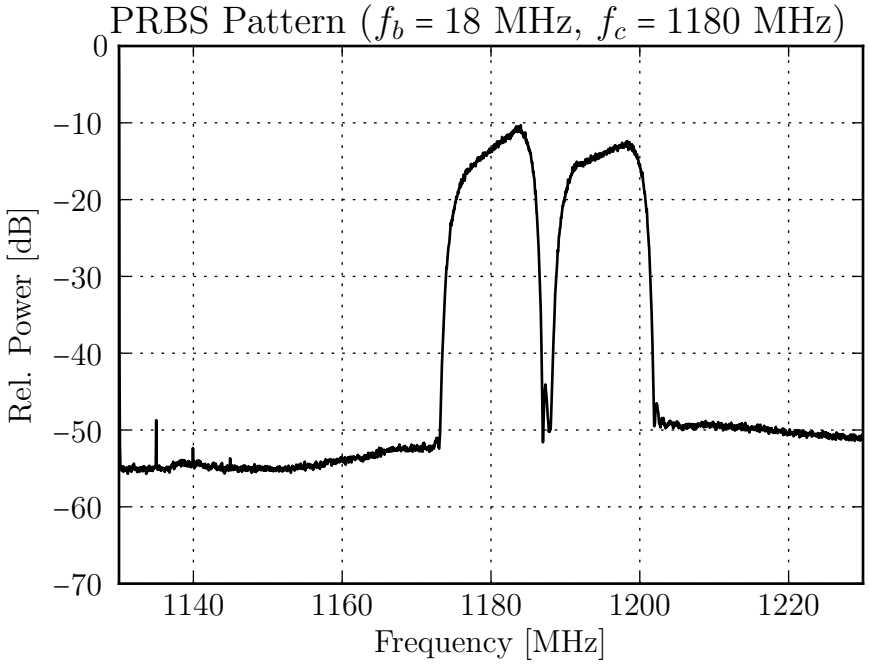


Figure 6-3: $f_c$ = **1180 MHz input waveform from AZ-410**. This portion of the spectrum is outside the target channel bandwidth and has severe aliasing problems, preventing the modem from operating correctly.

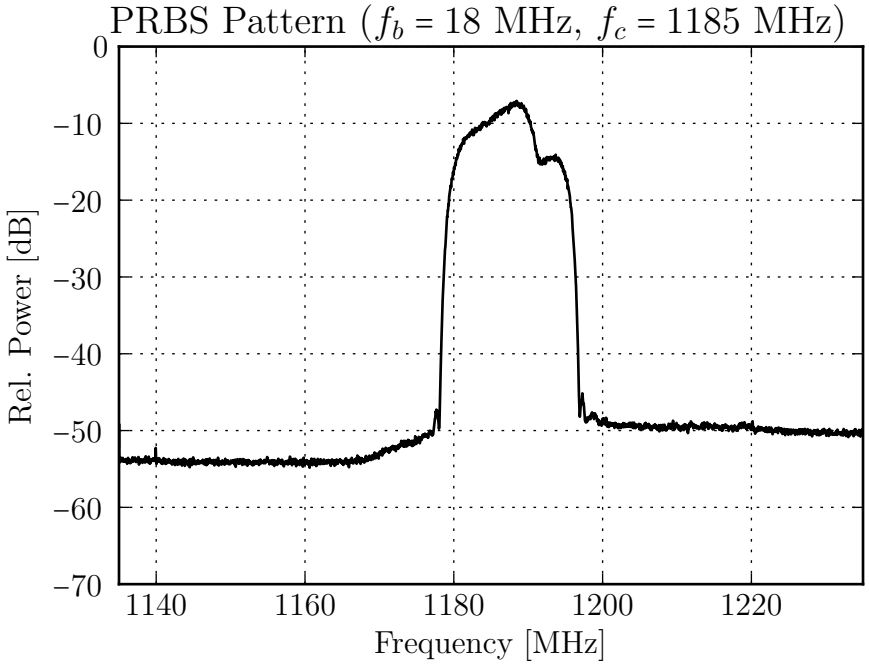


Figure 6-4: $f_c$ = **1185 MHz input waveform from AZ-410**. The aliased signal is folded into the main signal, also causing the modem to function improperly.

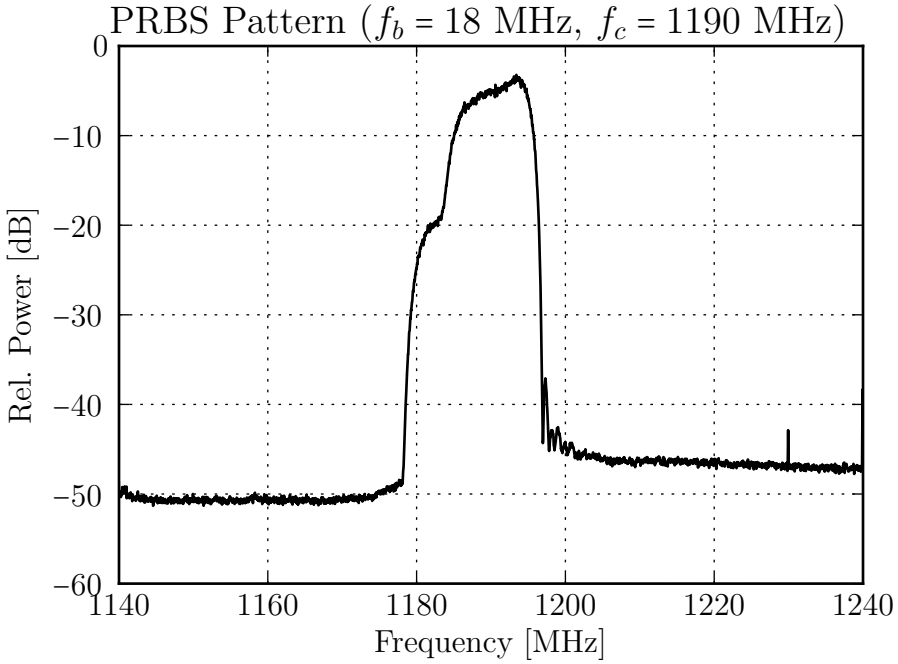


Figure 6-5: $f_c$ = **1190 MHz input waveform from AZ-410**. The aliased signal begins to emerge from the other side of the spectrum. At this point, the modem does in fact work correctly, though as Figure 6-2 indicates, the EVM was nearly 30%.

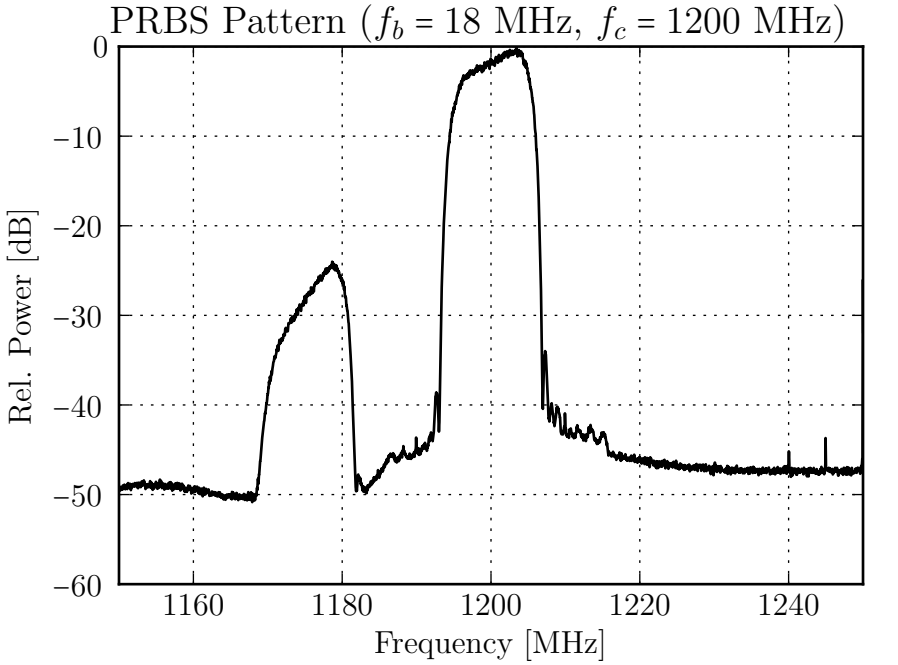


Figure 6-6: $f_c$ = **1200 MHz input waveform from AZ-410**. At this frequency, the EVM is low, below 20% and the spectrum is quite clean.

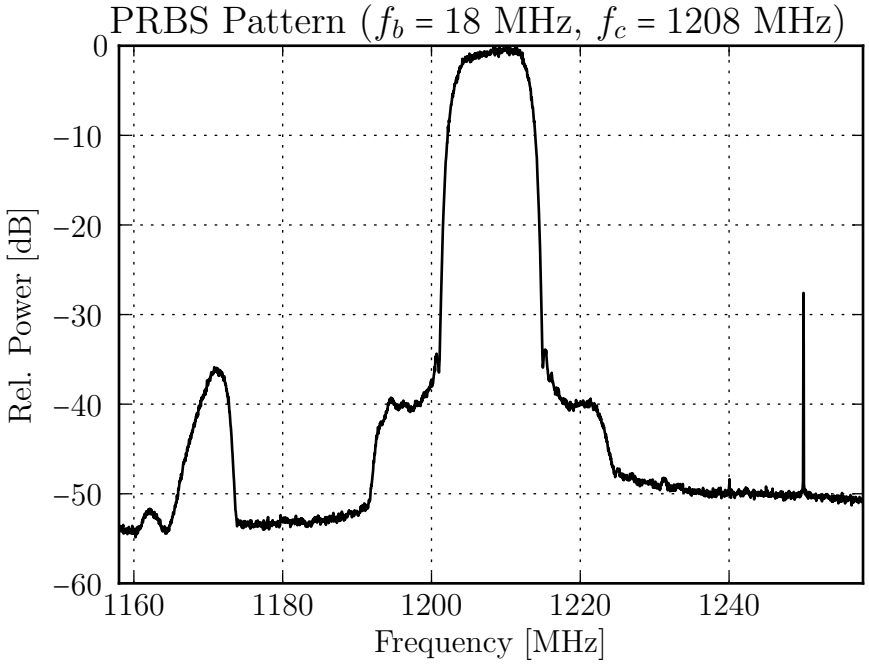


Figure 6-7: $f_c$ = **1208 MHz input waveform from AZ-410**. At $f_c$ = 1208 MHz, the aliased spectrum is over 35dB below the input spectrum, indicating that the channel emulator definitely performs to its 100 MHz target bandwidth spec on the lower side of the spectrum, though there is a small amount of roll-off.

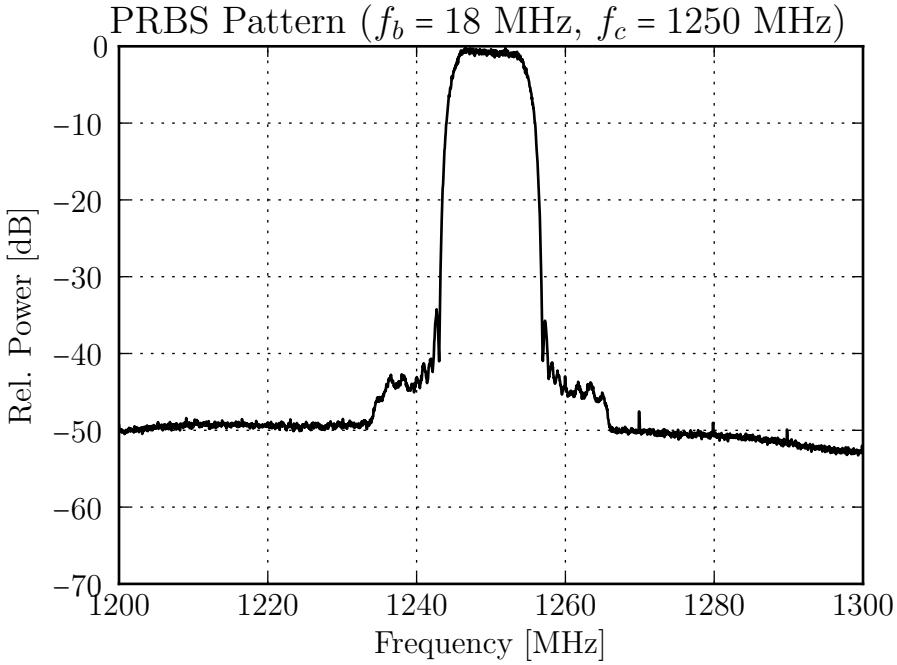


Figure 6-8: $f_c$ = **1250 MHz input waveform from AZ-410**. The spectrum is nicely symmetric about the center frequency. The best SNR achievable is around 50 dB, which by all industry best practices is excellent performance.

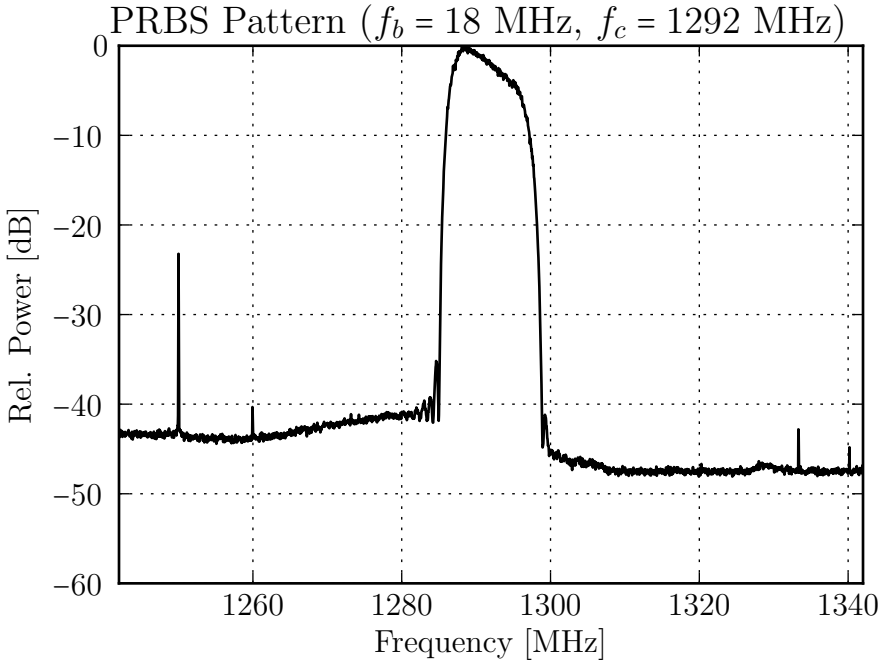


Figure 6-9: $f_c$ = **1292 MHz input waveform from AZ-410**. At the upper bound of the target channel bandwidth, there is some roll-off due to the analog low-pass filter at the output of the DAC (which was removed in later testing), but the signal quality is excellent. The EVM is just over 20%.

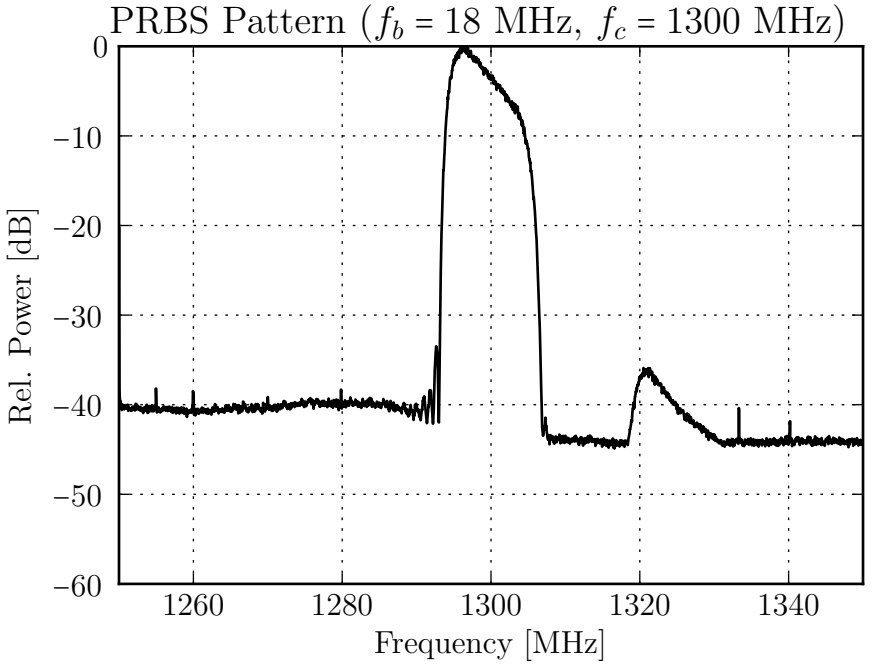


Figure 6-10: $f_c$ = **1300 MHz input waveform from AZ-410**. At a center frequency of 1300 MHz, an alias image begins to emerge, but is very small, so the modem continues to operate without issue.

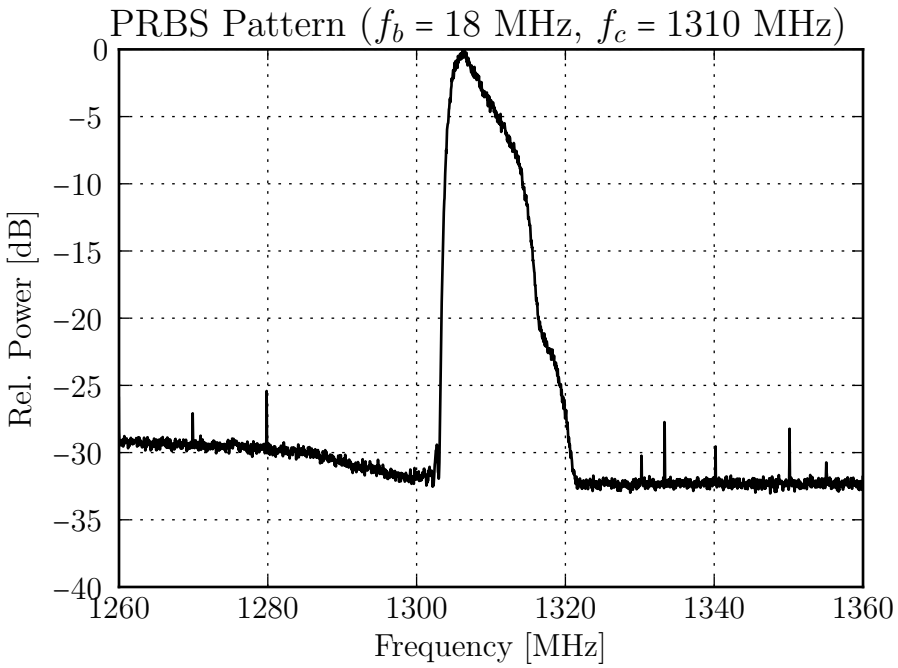


Figure 6-11: $f_c$ = **1310 MHz input waveform from AZ-410**. The alias image begins to merge with the primary signal, but the modem continues to function properly. This is essentially the farthest edge of the channel bandwidth.

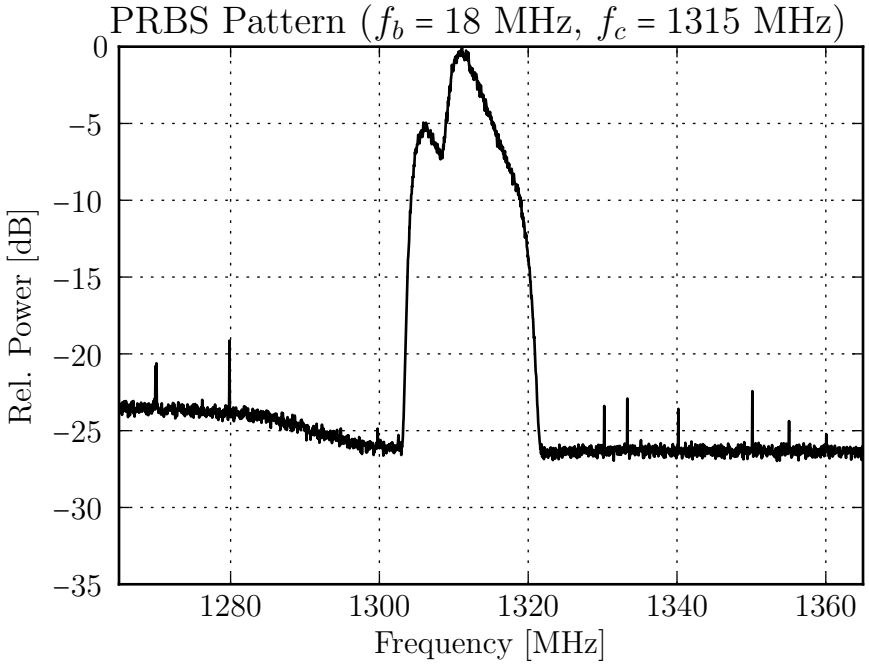


Figure 6-12: $f_c$ = **1315 MHz input waveform from AZ-410**. The alias image signal moves further left and the signal is so distorted that the modem ceases to function correctly.

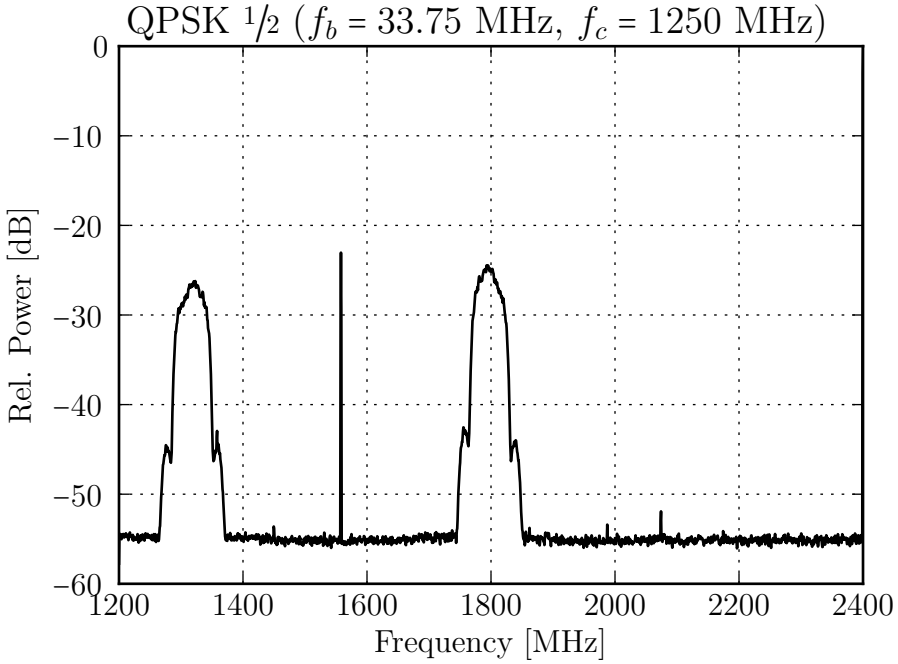


Figure 6-13: **25 MSPS input waveform from high-bandwidth modem**. At just under twice the bandwidth of the AZ-410 modem, the input spectrum is clean and the modem functions without issue.

spectrum just grazing the sampling frequency of the modem. As a consequence, there is inherently a spectral limit on this testing harness. Experiments with this setup concluded that the channel emulator was able to support a bandwidth of around 122 MHz, which is very well in line with the results of testing with the low-bandwidth modem.

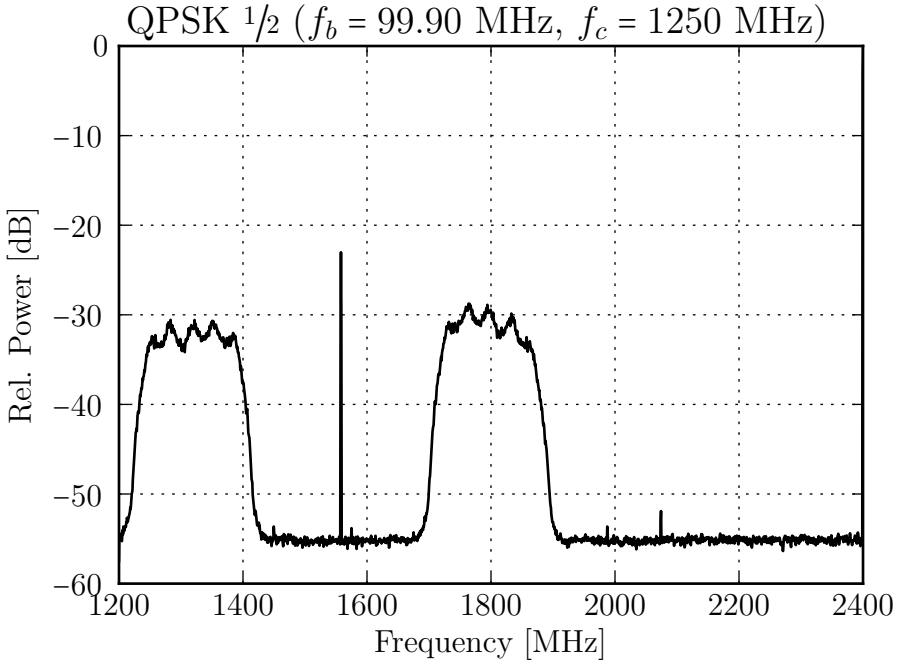


Figure 6-14: **74 MSPS input waveform from high-bandwidth modem**. At the target bandwidth, the modem continues to operate.

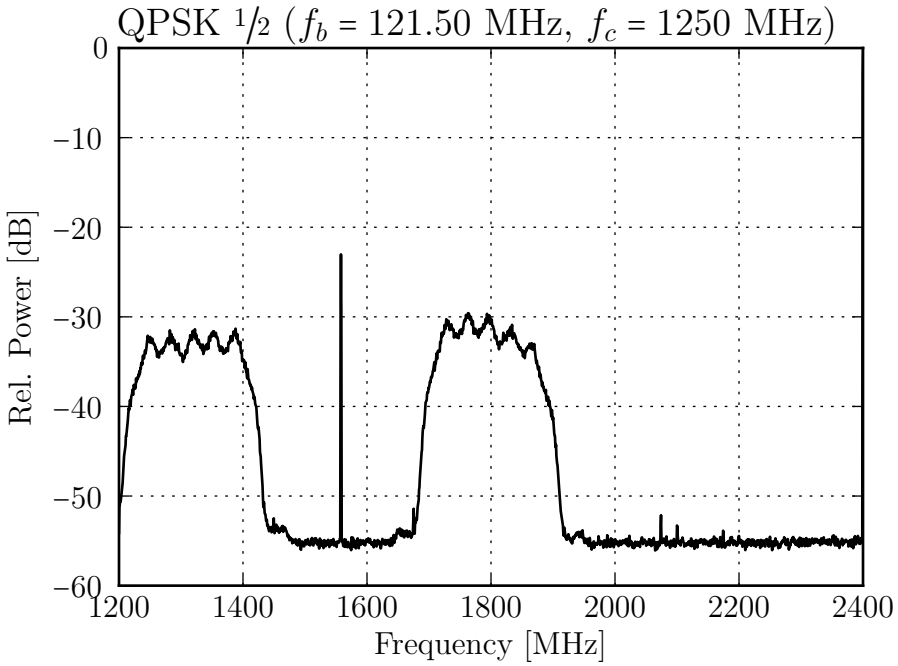


Figure 6-15: **90 MSPS input waveform from high-bandwidth modem**. At 121.5 MHz, the modem's sampling frequency begins to present an issue for the received spectrum, but the modem continues to function.

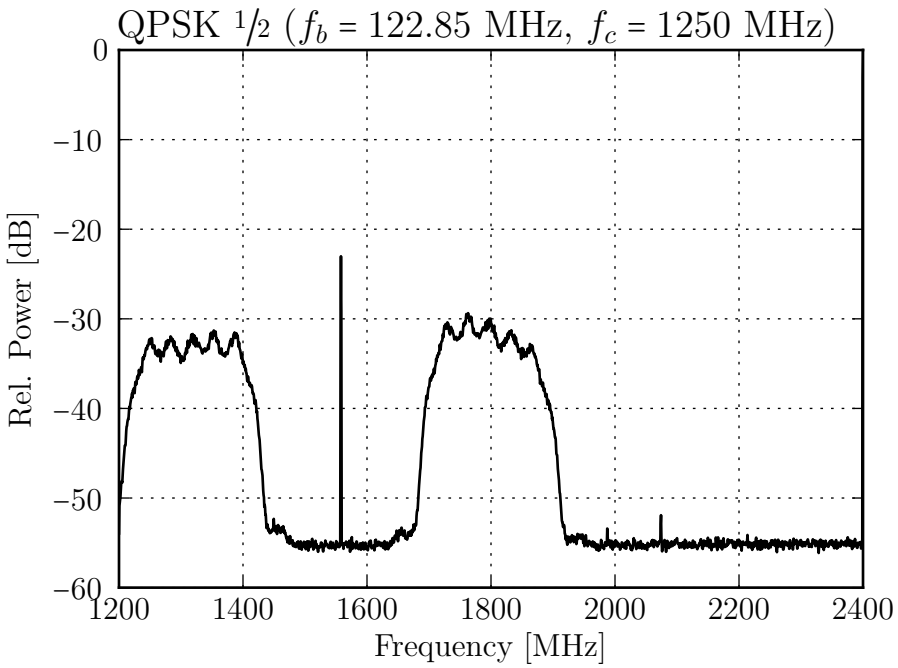


Figure 6-16: **91 MSPS input waveform from high-bandwidth modem**. At 122.85 MHz, the modem ceases to function. This result matches very closely to the perceived bandwidth limit of 125 MHz from the previous experiment.

# Chapter 7

# Testing Doppler Shifting Capabilities

Once the bandwidth limit of the emulator was measured, the emulator's Doppler shifting abilities were characterized and the AZ-410's resistance to Doppler shifting was tested.

After performing some initial tests, applying various Doppler profiles, it was determined that an excellent way to see the effect of Doppler shifting on a DVB-S2 signal is simply by looking at whether it is in sync or not because it was very difficult to construct situations where the signal was in sync but had a non-zero bit error rate.

The AZ-410 reports a variety of useful data via its administration console. First, it reports the offset from the center frequency. For example, if it reports an offset of +40 KHz, then the modem had to shift the spectrum to the *right* by 40 KHz to offset the frequency shift. Therefore, the spectrum itself had shifted to the *left* by 40 KHz. By simply inverting the sign of this reading, we have the measured frequency shift. The second useful piece of information it can return is whether or not the modem is in sync and receiving data. This data is continually extracted from the modem's configuration system via a script and the data is reported to `dopgen`'s logger.

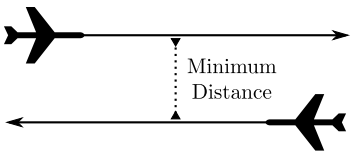


Figure 7-1: **Aircraft fly-by test path**. This elementary test path recreates the most classic Doppler experiment where a car playing a single tone travels by an observer. In this case, two aircraft fly by one another with some fixed minimum distances, as shown in the diagram.

## 7.1 Notes about Modem Data Capture

The AZ-410 has some fundamental design flaws that make it impossible to retrieve modem data in a timely manner. In order to capture data from the modem, a program continually makes requests to the modem's HTTP-based administration interface. Each request takes approximately one second. Furthermore, the modem itself only updates certain internal values, such as offset frequency, about once a second. So that the `dopgen` logger does not have to wait for requests, a buffering system is used in the request wrapper so that one thread is always requesting the latest position. When the logger asks for the modem status, the wrapper simply returns the last measurement it saw. As a consequence, the resulting modem data typically lags behind the commanded results (which are nearly instantaneous) by somewhere between half a second and a second. This will be evident in the observations below.

## 7.2 Constant Airspeed Fly-By

A very simple place to start is to consider a scenario where two planes fly by one another in opposite directions, as illustrated in Figure 7-1.

This scenario is much like the classic Doppler experiment where an observer stands still and listens to a car blaring its horn as it passes him. If the planes are travelling at constant airspeed, we expect that as they approach one another, there will be a constant positive Doppler shift. As they pass each other, the shift transitions from positive to negative. The speed of this transition is based on factors such as the planes' airspeeds and how closely they pass one another. Figures 7-2 and 7-3 represent

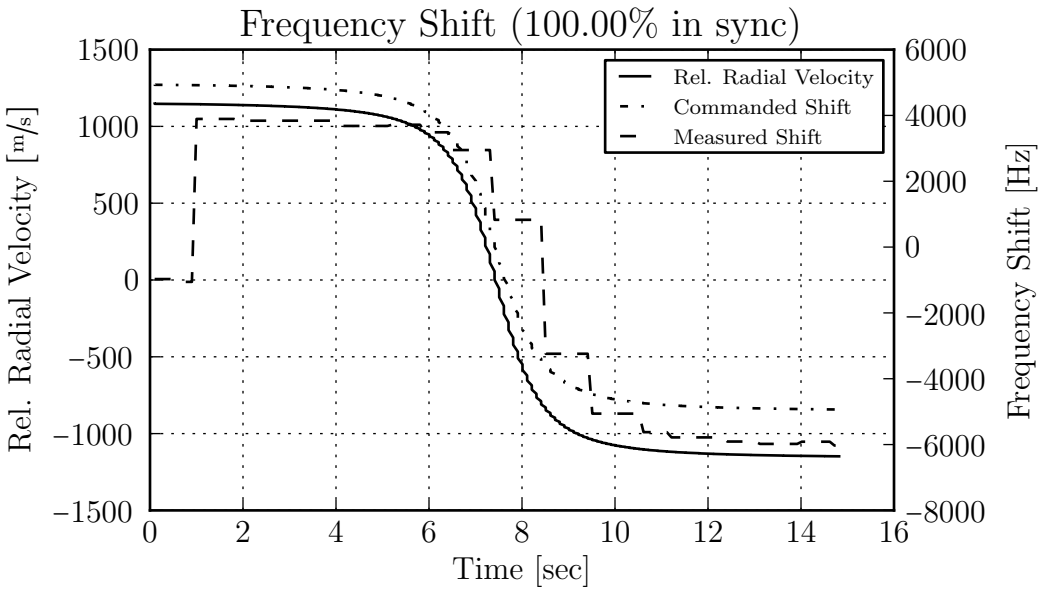


Figure 7-2: **Distant fly-by frequency shift**. Two planes fly by one another at a large distance, around 5000 ft.

scenarios where in both cases the planes are travelling at 1900 ft/sec, but in the first case, the closest separation is 5000 ft, and in the second case, the closest separation is 500 ft. Gray bands indicate that the modem has lost sync and is unable to deliver data. The transition time for the first case is slower and so the modem is more able to account for the Doppler shift. However, the extremely fast transition time in the second case causes the modem to fail at this point, as illustrated by the shaded area.

## 7.3 Flight Simulator Testing

Using X-Plane and `dopgen` in flight simulator mode, a single, relatively long (around 6 minutes) flight path was applied to a PRBS signal from the AZ-410, coded at 8PSK 5/6. The receiver was stationary on the ground, so the flight path was intended to approach and leave the area of the receiving antenna, which is located at 42.3768°N, −70.9994°W, as shown in Figure 7-4. The resulting frequency shifts are recorded in Figure 7-5. In areas of quickly changing shifts, the modem loses sync.

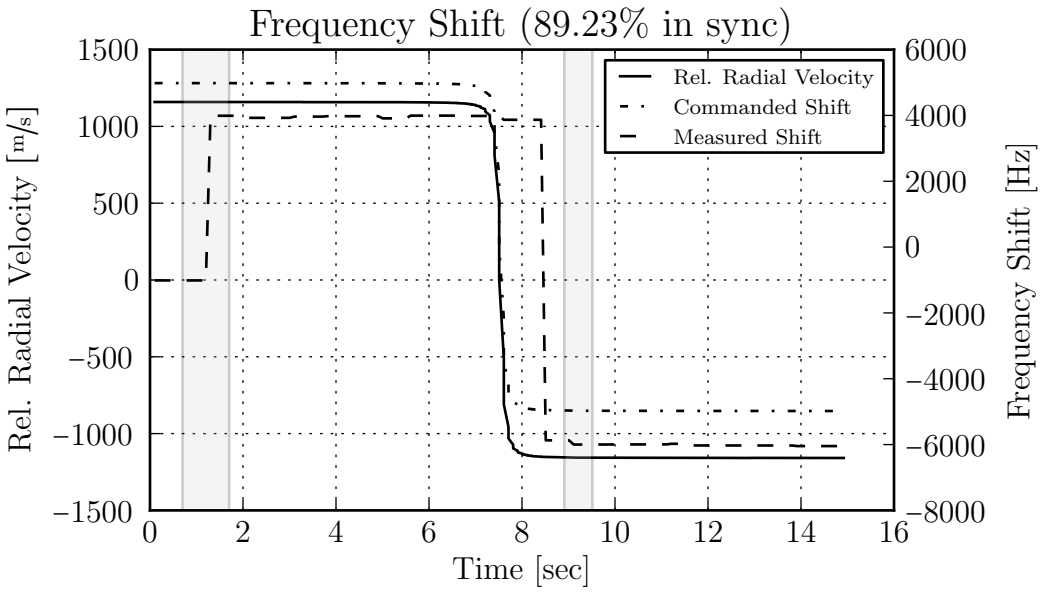


Figure 7-3: **Close fly-by frequency shift**. Two planes fly by one another closely, around 500 ft. The loss of communication means that the modem no longer successfully transmitted date 100% of the time.

## 7.4 Flight Simulator Merged Path Playback

Two flight simulator paths were recorded independently and merged into a single EEL file. The resulting merged path is shown in Figure 7-6. This path was then fed into `dopgen` multiple times in order to investigate consistency in applying a complex Doppler profile. One illustrative result is shown in Figure 7-7. Communication is lost when the frequency shift changes rapidly or is very large (around 3 KHz). After repeating this experiment multiple times, it was observed that the percentage of time the modem was successfully able to transmit data was between 80% and 90%.

Thus, the channel emulator makes it relatively straightforward to determine how various flight paths and the resulting frequency shifts from them affect the ability of the modem to receive a waveform.

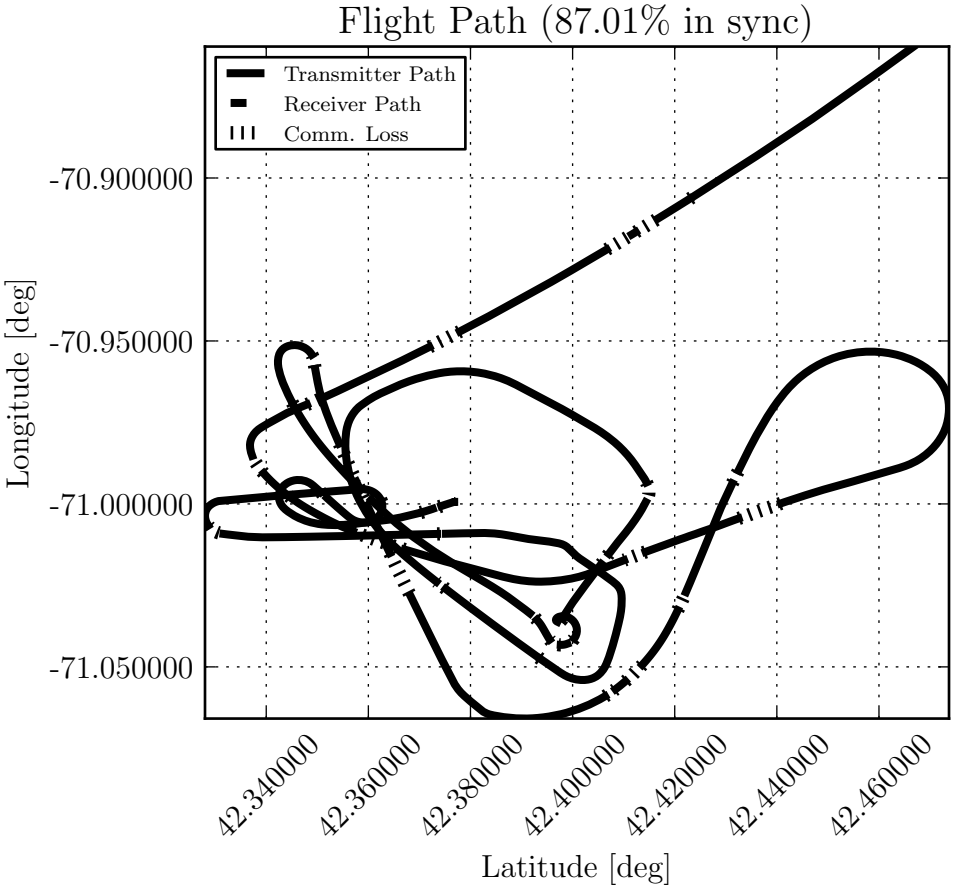


Figure 7-4: **Live system test frequency shift**. In this long, complicated flight test, the flight simulator fed transmitter coordinate data to `dopgen` and the modem's response was recorded. The receiver was a fixed point on the ground. The modem's received signal failed often during instances of quickly-changing frequency shifts.

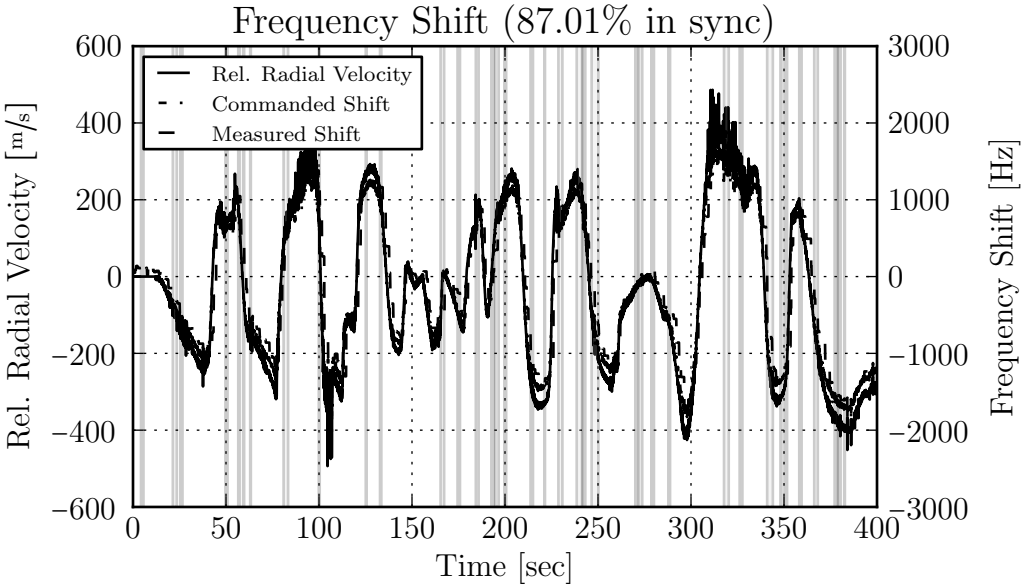


Figure 7-5: **Live system test frequency shift**. In this long, complicated flight test, the flight simulator fed transmitter coordinate data to `dopgen` and the modem's response was recorded. The receiver was a fixed point on the ground. The modem's received signal failed often during instances of quickly-changing frequency shifts.

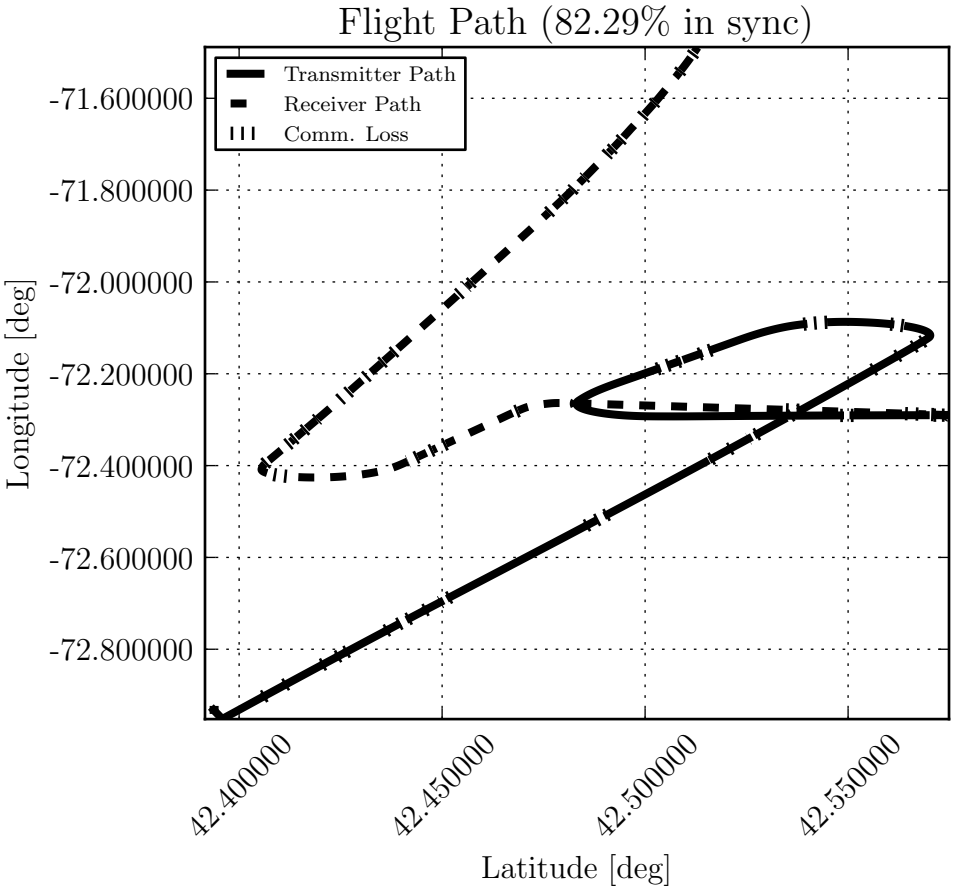


Figure 7-6: **Merged flight path frequency shift**. Two flight paths are merged into a single system as an example of a moving transmitter-receiver pair.

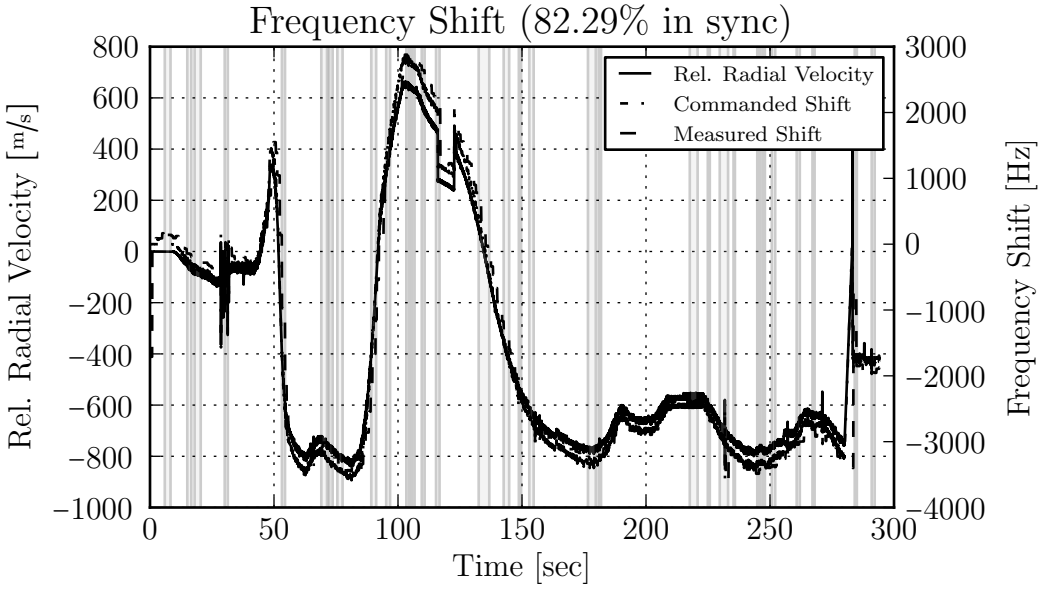


Figure 7-7: **Merged path frequency shift frequency shift**. Communication is very frequently lost in this run

# Chapter 8

# Concluding Remarks

The primary objective of building a scalable framework capable of applying channel emulation effects to high-bandwidth L-band signals was completed. This solution provides a solid design that allows for pluggable modules to be installed into the signal chain. It does so by handling all of the pre- and post-processing of the signal path and by encouraging signal modification at baseband.

A Doppler effect emulator, which shifts a spectrum based on the relative radial velocity of two aircraft was built. A fusion of techniques were used to realize this capability: the software program `dopgen` retrieves inputs from live or pre-recorded scenarios and directs the hardware to shift an input spectrum by varying degrees. The resulting solution was tested in a variety of environments and the results of that testing lead to the conclusion that satellite modems using DVBS-2 are not capable of correcting long periods of quickly-changing Doppler shifts.

## 8.1 Looking Forward

This project lends itself to an enormous number of opportunities for further development. With regard to the creation of new emulation modules, support for multipath signals, additive noise, and other features have already been identified as desired additional features. The existing framework should make the addition of these features relatively painless.

Outside of simply developing new modules, there are a variety of improvements that can be made to the underlying architecture for even more flexibility for future testing. First, replacing the $f_s/4$ DDC with a variable down-converter to support a wider variety of input center frequencies between 1 and 1.5 GHz. Second, by switching from a simple FPGA evaluation board to a custom board, a single FPGA can control input and output from multiple ADCs and DACs, as well as communicate with other FPGAs on the same board to allow emulation of very complex scenarios where multiple aircraft are communicating with one another in close proximity.

# Appendix A

# `cci` – Channel Emulator Configuration Interface

## A.1 Comments

Comments begin with `;`, which is the same comment character as that of PSM. There are no block comments.

## A.2 Inline Assembly

Inline assembly, which is directly injected into the generated assembly file, is surrounded by curly braces, with the braces each on their own line. For example, the following snippet would insert assembly code into the generated file:

```
label:
    load s0, AA
    call uart_tx
```

Note that the braces must be on their own line.

## A.3 Macro Reference

### A.3.1 `fmcw DEV XX YY` — SPI Write to FMC110

Writes `YY` to the address `XX` of the `DEV` chip on the FMC110 board, where `DEV` is one of `cpld` (master CPLD), `clkt` (clock generator), `adc0` (first ADC), `adc1` (second ADC), `dac0` (first DAC), or `dac1` (second DAC). If using `clkt`, then `XX` is a two byte value, with the high and low bytes separated by a space (e.g. `01 83`).

**Generated assembly (`DEV` ≠ `clkt`):**

```
load s0, XX
load s1, YY
call DEV_send
```

**Generated assembly (`DEV` = `clkt`):**

```
load s2, XX.hi
load s0, XX.lo
load s1, YY
call clkt_send
```

### A.3.2 `fmcr DEV XX` — SPI Read from FMC110

Reads from address `XX` of the `DEV` chip on the FMC110 board, where `DEV` is one of the devices listed in `fmcw`. If using `clkt`, then `XX` is a two byte value with bytes separated by a space, also as in `fmcw`.

**Generated assembly (`DEV` ≠ `clkt`):**

```
load s0, XX
call DEV_read
```

**Generated assembly (`DEV = clkt`):**

```
load s2, XX.hi
load s0, XX.lo
call clkt_read
```

### A.3.3 `fmcv DEV XX YY` — SPI Write and Verify from FMC110

Performs `fmcw` with data `YY` to address `XX` of device `DEV` and then performs `fmcr` on the same address, sending the resulting byte back via UART. This is useful for debugging to verify that the correct value was set.

**Generated assembly (`DEV ≠ clkt`):**

```
load s0, XX
load s1, YY
call DEV_send
call DEV_read
load s0, s1
call uart_tx
```

**Generated assembly (`DEV = clkt`):**

```
load s2, XX.hi
load s0, XX.lo
load s1, YY
call clkt_send
call clkt_read
load s0, s1
call uart_tx
```

### A.3.4 `ccfg DD AA` — FPGA Module Configuration

The CPU uses an auxiliary bus system to configure various modules. For example, through this system, one can set the digital output gain. This command will set address `AA` to have value `DD`. `AA`'s high nibble determines which module it configures and its low nibble determines which address of that module to configure.

**Generated assembly:**

```
outputk AA, port_cfg_addr
outputk DD, port_cfg_data
```

### A.3.5 `outk DD AA` — Shortcut for `outputk`

Shorthand for the `outputk` directive.

**Generated assembly:**

```
outputk DD, AA
```

### A.3.6 `out REG, PORT` — Shortcut for `output`

Shorthand for the `output` directive. `REG` can be `s0`, `s1`, etc.

**Generated assembly:**

```
output REG, PORT
```

### A.3.7 `in PORT` — Read from port

Reads from `PORT` and sends the value over the UART.

**Generated assembly:**

```
input s0, PORT
call uart_tx
```

### A.3.8 `sndstr "STRING"` — Sends a string over UART

Sends ASCII string over UART. Computes the MD5 hash of a random salt concatenated with `STRING` as a label and then uses PSM's iterative operation feature to send the string.

**Generated assembly:**

```
STRLABEL ← MD5_hex(random_salt + STRING)
string STRLABEL$, STRING
outputk STRLABEL$, port_uart_txdata
```

### A.3.9 `newline` — Sends a newline over UART

**Generated assembly:**

```
call send_newline
```

## A.4 Producing Assembly Code

`cci`'s default and most useful behavior is to produce assembly code which can then be assembled with PSM, the Picoblaze assembler. To generate code for an input file `infile.cci` to an output file `outfile.psm`, one can simply run the command:

```
sw/cci/cci.py -o outfile.psm infile.cci
```

As per UNIX standard, `-` can be used in place of `infile.cci` to represent reading from `stdin` and in place of `outfile.psm` to represent writing to `stdout`.

## A.5 Live Injection via Serial

An auxiliary useful feature of `cci` is that it allows the user to perform certain commands after the CPU has finished its initialization routine through the use of a prompt system.

The macros `fmcw`, `fmcr`, `fmcv`, `ccfg`, `outk`, and `in` can all be issued in this manner.

To issue a command over serial, the user runs `cci.py` with the `-s` switch specifying the serial port location and the location of a file containing a list of commands as normal. For example:

```
sw/cci/cci.py -s /dev/ttyUSB0 infile.cci
```

`stdin` can be particularly useful in this case. For example, the following shell command would issue the `fmcw` command to the system:

```
echo "fmcw adc0 01 00" | sw/cci/cci.py -s /dev/ttyUSB0 -
```

### A.5.1 Additional Options

In this mode, a few other options can be given. `-r` instructs `cci` to automatically read back the result from all `fmcw` commands to verify that the register was written. `-q` makes the output less verbose. If the command is a write command (i.e. `fmcw`, `ccfg`, or `outk`), the command returns nothing. If it's a read command (`fmcr`, `fmcv`, or `in`), it returns just the value read.

Finally, the `-c` switch can be used to specify a file containing constants. `cci` can read a subset of Picoblaze assembly to parse out constants used in the design and translate labels into numerical values. As a result, the user does not have to remember important numerical values.

### A.5.2 CPU Prompt Architecture

After initialization, the CPU jumps to a prompt routine. In this routine, it accepts and processes commands from `cci`'s serial processor.

To determine whether the prompt is available, `cci` starts by issuing the `?` command, seeing if it receives a `!` in response to indicate that the system is ready. It tries this 10 times before it fails and exists. After getting the prompt, it then translates each macro into a prompt command according to the protocol specified in Table A.1.

Table A.1: CPU Prompt Protocol. `<label>` represents a single byte, with the only exception being that `<addr>` is two bytes when writing to or reading from the clock generator.

| **Command** | **Description** |
|---|---|
| `? → !` | check if prompt is ready |
| `r<dev><addr> → <value>` | FMC110 SPI read. `<dev>` is `c` (CPLD), `C` (Clock Generator), `a` (ADC0), `A` (ADC1), `d` (DAC0), or `D` (DAC1). |
| `w<dev><addr><value>` | FMC110 SPI write. `<dev>` is as in the previous command. |
| `i<port> → <value>` | Read from input port. |
| `o<port><value>` | Write to output port. |

# Appendix B

# VHDL Preprocessor System

*Preprocessing* is a standard practice in the field of programming. A preprocessor provides a (commonly language-agnostic) method for modifying the input code before it is actually compiled. For example, the compiler GCC has an extensive preprocessor that enables the user to compile slightly different designs depending on configuration inputs he or she gives.

However, Xilinx's synthesis tools have no such capability. As a consequence of needing the ability to modify my design for different levels of debugging, GCC's preprocessor was re-purposed to operate on VHDL files through a script named `vhd-preproc`.

## B.1 Fooling GCC

The heart of `vhd-preproc` is the GCC command:

```
gcc -Dfoo=bar -Dbaz=buz -traditional-cpp -E -x c -P -C file.vhd
```

This invokes GCC to believe `file.vhd` is a C file and to only run the preprocessor on the file. It prints the output to `stdout`.

## B.2 Preprocessing the Project

The preprocessor determines which files to process using the `prj` file which Xilinx generates as a list of HDL files to synthesize.

The syntax of each line of the file is:

```
<language> <library> "<file>"
```

where `<language>` is `vhdl` and `<library>` is `work`. The `-p` switch determines the location of the project file to use.

A very simple `key:value` determines the appropriate configuration to use in preprocessing. The file contains pairs of `<label>=<definition>`, with each on its own line. These pairs determine the `-D` switches to GCC. `-c` specifies the location of the project file.

Finally, the `-P` switch specifies a prefix to apply to the generated file. For each VHDL file listed in the project file, `vhd-preproc` produces the output file with a new prefix. By default, for an input file named `file.vhd`, it will produce `_gen-file.vhd`.

Another way to use `vhd-preproc` is to have it clean out all of the generated files. This can be done with the `-C` switch.

One unfortunate design limitation is that XST will produce an error at the associated line number in the generated file instead of the original one, which makes debugging slightly more difficult.

# Bibliography

[1] Christian Doppler. *Über das farbige Licht der Doppelsterne und einiger anderer Gestirne des Himmels*. F. J. Studnlcka, 1842.